The Design and Implementation of SOLAR, a Portable Library for Scalable Out-of-Core Linear Algebra Computations

Sivan Toledo Fred G. Gustavson

IBM T.J. Watson Research Center

Abstract

SOLAR is a portable high-performance library for out-of-core dense matrix computations. It combines portability with high performance by using existing high-performance in-core subroutine libraries and by using an optimized matrix input-output library. SOLAR works on parallel computers, workstations, and personal computers. It supports in-core computations on both shared-memory and distributed-memory machines, and its matrix input-output library supports both conventional I/O interfaces and parallel I/O interfaces. This paper discusses the overall design of SOLAR, its interfaces, and the design of several important subroutines. Experimental results show that SOLAR can factor on a single workstation an out-of-core positive-definite symmetric matrix at a rate exceeding 215 Mflops, and an out-of-core general matrix at a rate exceeding 195 Mflops. Less than 16% of the running time is spent on I/O in these computations. These results indicate that SOLAR's portability does not compromise its performance. We expect that the combination of portability, modularity, and the use of a high-level I/O interface will make the library an important platform for research on out-of-core algorithms and on parallel I/O.

1 Introduction

This paper describes the design and implementation of SOLAR, a high-performance portable library for out-of-core dense matrix computations. SOLAR is designed to meet three main objectives. First, the library should deliver as much of the functionality of LAPACK [2], a public domain library for in-core matrix computations, as possible. Second, the library should be portable across a wide variety of architectures. Third, the library should deliver high performance. The paper explains how the design allows us to achieve our objectives, and demonstrates that the implementation is indeed portable and achieves high performance. Our current implementation does not yet include the full functionality of LAPACK for out-of-core matrices, but it does include both sequential and parallel solvers for general and positive-definite symmetric linear systems as well as several support routines.

Although current computers, especially parallel computers, have large amounts of memory, there is still a need for software for out-of-core matrix computations. Since DRAM is about 100 times more expensive than disk [18] and since out-of-core software for dense matrix computations can run at almost the rate of in-core software (see Section 4), out-of-core software can solve problems at a much lower cost-performance ratio than in-core software. Out-of-core solvers can be used for the overnight solution of very large problems on workstations, for the solution of large problems in easy-to-use modeling environments such as Matlab, as well as for solving huge problems that do not fit within the primary memory of any existing computer. A testimony for the need for out-of-core software is the steady stream of implementations of such codes over the last 45 years [3, 4, 11, 12, 13, 14, 15, 20, 21, 19, 22, 25], including a number of recent implementations (for example, [4, 12, 13, 15, 25]). Unfortunately, none of these codes appears to be portable, and all of them provide only a few subroutines that the implementors needed, rather than a full set of dense solvers. Consequently, many users cannot take advantage of these codes.

Ubiquitous parallel computing in the form of scalable parallel computers, symmetric multiprocessors and networks of workstations provides two new incentives to use out-of-core algorithms. First, it is often more cost-effective to speed up a computation that does not fit in core by adding processing units and continuing to use an out-of-core solver than by than by adding enough memory to run it in core. Second, it is now possible to increase the performance of the I/O system using parallel I/O with disk arrays and parallel file systems.

But parallel computing and parallel I/O also pose new challenges to out-of-core software. Current computer systems offer many different computing and I/O environments. The processing unit can be a uniprocessor, a symmetric multiprocessor with a shared memory, or a parallel computer with a distributed memory. The I/O can be performed by a conventional file system or a parallel file system, and some of these file systems can use disk arrays or disks installed on several server nodes. A portable out-of-core library that aims to support all of these environments should therefore use flexible interfaces that can work in all of these environments. The interfaces should also enable the use of multiple alternative external software modules, such as multiple file systems and multiple in-core subroutine libraries.

Besides providing users and application developers with a much-needed functionality, SOLAR will serve as a research tool for two communities. The authors plan to use the library in research on out-of-core numerical algorithms, and we hope that others will use it for research in algorithms as well. The history of linear algebra software indicates that libraries that support new architectures often advance the state of the art. We expect novel developments to emerge from SOLAR as well. Researchers working on I/O issues, such as parallel file systems and I/O-subsystem architecture will be able to use the library for empirical performance evaluations. Using a full-featured library as a benchmark is preferable to using a "toy" application since it is more likely to represent real workloads. We believe that the main reason for using toy applications in past research on I/O has been the lack of portable out-of-core software, and our library should eliminate this problem. The high-level I/O interface that SOLAR uses is compatible with current research on parallel I/O (see [16, 23], for example) and the library should therefore provide a convenient platform for research on I/O optimization.[1]

Our library uses two main mechanisms to achieves our design objectives:

1. The library combines high performance with portability by separating the generic software that can be used on any machine from a small set of functions that should be reimplemented by vendors to achieve high performance on their platforms. The platform dependent layer of the library consists of a set of functions that perform I/O on submatrices, as well as of high-performance in-core subroutines that are already implemented on most platforms. The in-core subroutines consist of the so-called Basic Linear Algebra Subroutines [9, 10, 17] (BLAS), the parallel BLAS [5], LAPACK [2], and ScaLAPACK [6]. The generic components of the library consist of subroutines that schedule I/O operations and call the in-core subroutines to manipulate submatrices stored in in-core buffers.

2. The library will provide a large set of solvers by following a traditional layered approach to linear algebra software. In this approach, which is used by LAPACK and ScaLAPACK, a large set of solvers is implemented in terms of a small set of BLAS. Our library uses BLAS that can operate on matrices stored in or out of core. These BLAS speed up the construction of solvers in two ways. First, it is often relatively easy to implement a solver using only calls to the BLAS. Second, since LAPACK already uses this approach, it is easy to translate the in-core subroutines in LAPACK into new out-of-core subroutines.

The rest of the paper describes the design and implementation of SOLAR and its performance and portability records. Section 2 describes the overall design of the library and its internal and external interfaces. Section 3 describes design decisions that were made during the detailed design and implementation of individual modules and subroutines. Section 4 presents the performance and portability records of SOLAR. Performance results show that SOLAR achieves high performance by efficiently utilizing both the processor or processors and the available I/O bandwidth. A porting experiment shows that SOLAR is portable. Our conclusions are presented in Section 5.

2 Software Organization and Interfaces

This section describes the overall design of the SOLAR and some of its internal and external interfaces. The discussion focuses on the interaction between the design of the interfaces and the performance of the resulting software. In particular, we describe the specification of matrices stored in core and out of core, the specification of I/O operations, the interaction between internal interfaces and software pipelining, and the choice of external interfaces.

SOLAR consists of three main modules: subroutines for out-of-core computations, interface subroutines for in-core computations, and subroutines for input-output operations on submatrices. The role of the out-of-core subroutines, both solvers and BLAS, is to schedule input-output operations and to manage the in-core memory. These subroutines must store and load submatrices to and from secondary memory and anticipate the need for submatrices during the computation so that they can be prefetched. The role of the interface subroutines for in-core computations is to allow a single out-of-core subroutine to work on several kinds of matrices by providing runtime type resolution. For example, the in-core interface subroutine for matrix multiply-add calls a sequential or parallel subroutine for real or complex matrix multiply-add in single or double precision based on runtime type information. The role of the matrix input-output subroutines (MIOS), is to read and write submatrices from and to secondary storage. We expect that vendors will implement versions of the MIOS optimized to their platforms.

Matrix Storage and the Matrix Input-Output Subroutines (MIOS)

SOLAR uses two forms of matrix storage, in-core and out-of-core. A matrix, whether in-core or out-of-core, is described by a matrix descriptor that specifies its data mapping and the type of its elements. A set of subroutines known as the matrix I/O subroutines is responsible for transferring submatrices of out-of-core matrices to and from main memory.

In-core matrices come in two main flavors: shared-memory matrices that can be manipulated by the sequential BLAS and by LAPACK, and distributed matrices that can be manipulated by the parallel BLAS and by ScaLAPACK. The library design completely specifies the structure of an in-core matrix descriptor. This specification is used by in-core subroutines, by the MIOS to determine the address of specific matrix elements, and by user programs when they access blocks of out-of-core matrices.

Out-of-core matrices, on the other hand, can be stored in any manner that the input-output module supports. While one MIOS implementation might support only matrices laid out in column major order in a conventional linear file, another MIOS implementation might support matrices laid out in blocks on multiple disks in multiple I/O nodes. There are two reasons to allow such diversity. First, it seems too early to decide exactly which out-of-core data mappings are necessary to achieve high performance in every possible dense matrix computation. Second, the specification of out-of-core data mappings depends on how I/O requests and file structures are specified to the operating system or to the file system, and it seems that too early to settle this issue [8]. Fortunately, it is possible to allow such diversity, because only the MIOS need to know the details of the out-of-core data mapping. Hence, the out-of-core storage descriptor can be viewed as an opaque object accessible only to the MIOS. Information contained in the descriptor about the matrix and its layout may be obtained by calling a MIOS inquiry function. In contrast, it is not possible to declare the in-core matrix descriptor an opaque object, because the data mapping of in-core matrices is used not only by the MIOS, but also by the in-core subroutines and by user programs.

The MIOS provide an I/O interface for user programs and for the out-of-core subroutines. The services that the interface provides include construction of descriptors, file open and close operations, asynchronous reads and writes of submatrices, and inquiry subroutines. There are several reasons to perform I/O in out-of-core subroutines through a matrix interface. First, such an interface allows an out-of-core subroutine to use multiple out-of-core storage layouts and multiple low-level I/O interfaces without explicit reference to all of these options. Second, a high level interface allows for I/O optimizations such as two-phase I/O [23] and disk-directed I/O [16].

The contract between the out-of-core subroutines and the MIOS is simple. The MIOS can transfer any rectangular block of a matrix between primary and secondary memories. To enable the out-of-core subroutines to request data-transfers that efficiently utilize the I/O bandwidth of the machine, the MIOS provide an inquiry subroutine that returns the dimensions of a **primary block**, the basic unit in which an out-of-core matrix is stored. Data transfers of aligned

[1] SOLAR's source code is freely available from the authors for research purposes. To receive a copy, send email to `toledo@watson.ibm.com`.

1	2	3	4	5	6	1	2	3	4	5	6
4	5	6	1	2	3	4	5	6	1	2	3
2	3	1	5	6	4	2	3	1	5	6	4
5	6	4	2	3	1	5	6	4	2	3	1
3	1	2	6	4	5	3	1	2	6	4	5
6	4	5	3	1	2	6	4	5	3	1	2

Figure 1: A 2-dimensional cyclically-shifted block layout of a 6-by-12 block matrix on a 2-by-3 disk grid. Each square in the figure represents a block of the matrix. The digit in each block specifies which disk owns the block. Any aligned block of size 2-by-3, 1-by-6, or 6-by-1 is evenly spread across the six disks.

and contiguous primary blocks should yield the best possible I/O bandwidth. Thus, even though the out-of-core subroutines are not designed with any specific data mapping in mind, they can achieve high performance by using only aligned contiguous primary blocks in data transfers. User programs whose performance is not critical, on the other hand, can ignore the issue of primary blocks and transfer submatrices of any shape. The primary block of an out-of-core matrix is selected when the matrix is created. Choosing a large primary block reduces the amount of time spent on moving the disk arm (seeking) between primary blocks, but restricts the choice of a submatrix whose transfer is efficient.

When a matrix is striped across several disks that allow independent I/O (that is, access to a different block on each disk) each disk stores one or more primary blocks. In such cases, loading or storing a combination of primary blocks that load balances the disks yields higher I/O bandwidth than a combination in which the load is unbalanced. Although we can simply stripe each primary block across the disks, this leads to larger primary blocks and less flexibility. This flexibility can be valuable when different parts of an algorithm use submatrices of different shapes. In LU decomposition with partial pivoting, for example, factoring column blocks require reading a group of entire columns, but I/O of square blocks is best for the update of the trailing submatrix.

We have discovered a distributed matrix layout that supports efficient I/O transfers of submatrices of several important shapes. The layout, called the **2-dimensional cyclically-shifted block layout**, is illustrated in Figure 1. Using this layout, an array of r-by-c disks stores a matrix such that each aligned r-by-c, rc-by-1, and 1-by-rc group of primary blocks is striped across all the disks. Such a layout allows for efficient transfers of square or roughly square submatrices as well as blocks of rows or columns. This layout is scalable in the sense that all three block shapes are possible with a block size that is only rc, the number of disks. A conventional block-cyclic layout using an r-by-c disk grid might require loading more than rc primary blocks to get an even load balance. SOLAR now uses a block-cyclic layout for laying out matrices on parallel disks, but we plan to support the 2-dimensional cyclically-shifted block layout in the future.

We present this new layout in order to demonstrate that it is not wise to fix at this point the types of out-of-core matrix layouts that the MIOS module should support. Since yet undiscovered layouts may prove superior to conventional block-cyclic layouts, and because it is not necessary to specify the layout in most of the MIOS module's interfaces (all except matrix creation), we leave the layout of out-of-core unspecified in the specification of the MIOS.

Since the MIOS interface does not specify all the possible data mappings for out-of-core matrices, the arguments to the descriptor construction subroutine are not specified by the interface, and an implementation can use any set of arguments. Likewise, the arguments to the file open subroutines are implementation dependent. The main implication of the design decision to leave the arguments to these subroutines unspecified is that out-of-core subroutines in the library cannot create temporary files in a portable way. Since LAPACK does not allocate memory either, but relies instead on work areas supplied by the caller, we felt that this restriction is not significant. Where needed, our library can rely on work files supplied by the caller without deviating from the calling sequences of LAPACK.

Pipelining Policies

SOLAR uses an asynchronous I/O interface in order to overlap I/O and computation. The asynchronous I/O interface is used to pipeline data to and from secondary memory. A subroutine that needs to read a matrix from a file, for example, issues the request to read block $i+1$ (a so-called **prefetch**), waits for the completion of the reading of block i, then computes on block i. The reading of block $i+1$ overlaps the computation performed on block i. This 2-stage pipeline requires two buffers per matrix. If the blocks are modified and must be written back to secondary memory, then the writing of block $i-1$ overlaps the computation performed on block i, and the 3-stage pipeline requires three buffers per matrix.

Our design restricts the use of asynchronous I/O using two simple policies. The first policy is that each subroutine must perform its own I/O, including prefetching, and each subroutine must wait until the data it writes has been accepted by the I/O system. Since a caller cannot perform the first prefetching on behalf of a callee or wait for its writes to complete, calling an out-of-core subroutine incurs a pipeline startup cost and a pipeline shutdown cost.

The second policy is that a matrix argument is always passed to a subroutine completely in core or completely out of core. In other words, a subroutine that receives an out-of-core matrix A as an argument cannot use a buffer that contains a submatrix of A if the buffer was read before the invocation of the subroutine, or if the buffer was computed in core before the subroutine was invoked. The subroutine must issue read requests for the entire matrix, even though it might mean that the program writes a buffer and then immediately reads it again or reads the same submatrix twice in a row.

The two policies keep internal interfaces in SOLAR simple and keep the implementations of different subroutines independent. If a subroutine is allowed to prefetch on behalf of another, then the first must use some assumptions on the implementation of the second. Interfaces are kept simple because pending I/O requests, active buffers, and matrices that are partially in-core are never passed as arguments to subroutines.

Our pipelining policies, whose goal is to keep SOLAR's design simple and modular, do have several implications on the performance of out-of-core subroutines. First, the performance of an iterative[2] block algorithm can be better than the performance of a recursive algorithm, since the recursive code incurs a pipeline startup cost after every recursive call, whereas the iterative algorithm incurs this cost only once. Second, increasing the block size in block algorithms increases the cost of the pipeline startup. In level-3 algorithms, such as matrix-matrix multiplication, it is best to choose a block size that is large enough to hide the I/O latency but not much larger than that. Our implementation takes these issues into account. While it might be argued that using the BLAS also introduces some overhead, since

[2] We use the term **iterative** in this paper to describe algorithms that span an iteration space using nested loops rather than using recursion. This use of the term should not be confused with its use to describe the solution of numerical problems by successive approximations.

each time a BLAS is called it must start its pipeline, we felt that this is a small penalty compared to the advantages that the BLAS offer. The advantages that the BLAS offer include modularity and ease of conversion of in-core software that uses the BLAS into out-of-core software. The last advantage is especially important since LAPACK's use of the BLAS is designed to enhance data locality on machines with caches, and such a design is naturally suited to out-of-core algorithms.

External Interfaces

The main design goal of the external interface of SOLAR is compatibility with the BLAS, PBLAS, LAPACK, and ScaLAPACK. The rationale behind this decision is to simplify the adaptation of user code to out-of-core matrices as well as to simplify the translation of LAPACK's solvers into out-of-core solvers. One superficial difference between existing libraries and SOLAR is the polymorphism in SOLAR. A single subroutine implements a given function for all data types instead of eight in existing libraries (four sequential and four parallel for single, double, complex, and double-complex data types). Since out-of-core computations are typically time consuming, the overhead caused by runtime type resolution is insignificant.

We have therefore decided that the calling sequences of SOLAR should be identical to the calling sequences of the parallel BLAS and ScaLAPACK. In the calling sequences, a matrix or a submatrix is specified by four arguments: the address of the beginning of an array, two integer indices, and an integer array containing a matrix descriptor. In ScaLAPACK and the parallel BLAS, the descriptor describes the layout of a distributed matrix, the array contains the the portion of the matrix that is stored on the processor, and the two indices determine the row and column in the matrix where the specified submatrix begins. In SOLAR, the descriptor describes either an out-of-core matrix or an in-core matrix. An in-core matrix can be stored in shared memory (or the memory of a uniprocessor) or in distributed memory, in which case the SOLAR descriptor contains a ScaLAPACK descriptor. The indices determine the beginning of the specified submatrix. The array whose address is passed contains the local portion of the in-core matrix when the descriptor indicates that the matrix is indeed in core, and it is not used at all when the matrix is out of core.

It is legal to pass to a subroutine with two or more matrix arguments some arguments in core and other arguments out of core. A solver, for example, can be called with an out-of-core coefficient matrix and a single in-core right hand side in some cases and with an out-of-core coefficient matrix and an out-of-core matrix of multiple right-hand sides in other cases. A Fortran program that uses SOLAR to solve a large linear system is shown in Figure 2.

3 Detailed Design and Implementation

We have so far implemented the matrix I/O subroutines module, three of the level-3 BLAS (each for four elementary data types), and factor and solve subroutines for general as well as positive-definite symmetric matrices. The MIOS module supports both conventional I/O and parallel I/O and supports in-core matrices stored in block-cyclic distributed layouts. All the computational subroutines except the general LU factorization work in both sequential and parallel environments. The LU factorization currently only has a sequential version and we are working on a parallel version.

This section describes the current implementation of SOLAR. The section focuses on the most important decisions that were made during its design and implementation. We start by describing the process by which we chose the source implementation language. Although the choice of a language is inessential in many projects, we feel that our decision to use C requires some justification given the excellent portability record of numerical software written in Fortran and the capabilities of Fortran 90. We then turn to the algorithmic design issues that arose during the design of the out-of-core subroutines in SOLAR. The section concludes with a description of the MIOS module.

Source Language

After weighing several factors and implementing one subroutine in C, Fortran 77, and Fortran 90 we decided to implement SOLAR mostly C in order to simplify the use of pointers. Fortran 77 subroutines can be incorporated into the library, and the in-core components that we use are written entirely (BLAS and LAPACK) or partially (PBLAS and ScaLAPACK) in Fortran 77. The library is callable from Fortran 77 and all our test programs are written in Fortran.

The deciding factor to use mostly C was the ability to alias arrays conveniently in C, a feature missing from both Fortran variants. Consider a two-stage pipelining in an our-of-core subroutine. When the subroutine computes on a submatrix stored in an in-core buffer another submatrix is read into a second buffer. When the computation and the read operation terminate, the submatrix that was just read moves to the compute stage. It is inefficient, however, to copy it to the other in-core buffer. It is more efficient to use the buffer where that submatrix is stored as the compute buffer in the next stage in the algorithm and to read a new submatrix into the buffer that was previously used for computation. This strategy can be easily realized by defining two in-core buffer and two pointers to such buffers. One pointer always points to the compute buffer and the other always points to the read buffer. When a stage in the algorithm ends, the pointers are swapped.

Another situation in which aliasing is important occurs when an out-of-core subroutine passes an in-core buffer to an in-core subroutine. In SOLAR, the in-core buffer can be either an array argument of the out-of-core subroutine or a dynamically declared array if the argument was an out-of-core matrix. If we wish to have a single call to the in-core subroutine, then we must pass to it a pointer that points to either one of the two arrays. Using a separate call for each case makes coding awkward when the call passes two or three such arguments.

It is easy to implement aliasing strategies in C using pointers. It is impossible to implement such strategies in Fortran 77 if the in-core buffers are not allocated as a single array. Fortran 90 supports pointers, but they cannot point to subroutine arguments that are assumed-sized arrays. Therefore the argument cannot be passed by a Fortran 77 or C program, but only by a Fortran 90 program. Since it is important that the SOLAR can be called from Fortran 77 and C, we could not use Fortran 90 pointers.

Implementing SOLAR in C causes two difficulties. First, it is more time consuming to convert a in-core Fortran subroutine from LAPACK into an out-of-core C subroutine than it is to convert the in-core code into a Fortran out-of-core subroutine. Second, C lacks complex data types, and the correspondence between real (and integer) data types in Fortran and in C is machine dependent. To circumvent this potential portability problem we developed a small module with data-type definitions and auxiliary functions that isolate all the dependencies on the Fortran-to-C data-type mappings. For example, this module defines `I_type` to be the C data type that correspond to the Fortran `INTEGER` data type. This module also allows SOLAR to work on complex data. SOLAR creates a complex variable with the value 1, for example, by calling a function that returns a pointer to a complex datum with a given initial value, `F77types_get_value("C",1.0)`.

Subroutines for Out-of-Core Computations

We have already implemented three out of the nine level-3 BLAS [9], namely general matrix multiply-add, symmetric rank-k update, and triangular solve, as well as factor and solve subroutines for positive-definite symmetric and general matrices.

The BLAS use block-iterative algorithms. As explained above, an iterative block algorithm starts its pipeline only once, whereas a

```
      program solar_example

      integer   R,C                   ! Number of rows and columns in the out-of-core matrix A.
      parameter (R=12288,C=12288)

      integer   BR,BC                 ! Number of rows and columns in the in-core submatrix of A.
      parameter (BR=1024,BC=1024)

      double precision Aij(BR,BC)     ! An in-core buffer that will store a submatrix of A.
      double precision v(C)           ! An in-core column vector.
      integer          ipiv(C)        ! Vector of pivot indices.

      integer          A_desc(64)     ! Out-of-core matrix descriptor.
      integer          Aij_desc(64)   ! In-core submatrix descriptor.
      integer          v_desc(64)     ! An in-core descriptor for v.

      integer          request(8)     ! Asynchronous request descriptor.

      integer          i,j,info       ! Indices and error information.

      call mios_init()                ! Initialize the MIOS module.

      call mios_make_desc_ooc_aio(A_desc,'D',              ! Create a descriptor for an out-of-core
     $       128,128,R/128)                                ! double-precision matrix with 128-by-128
                                                           ! primary block and R/128 blocks per column.

      call mios_open_many_aio(A_desc,1,2,1,                ! Open a matrix stored on a 2-by-1 disk
     $       '/dev/hdisk0\0',0,                            ! Grid. The matrix starts at offset 0 on
     $       '/dev/hdisk1\0',0)                            ! both disks.

      call mios_make_desc_ic_shared(Aij_desc,'D',BR,BC,0)  ! Create 2 in-core matrix descriptors.
      call mios_make_desc_ic_shared(v_desc,'D',C,1,0)      ! Both are marked by 0 as non-dedicated.

      do j=1,C,BC
         do i=1,R,BR
c
c           Fill in Aij with the submatrix of A starting in location (i,j) (omitted).
c
            call mios_iwrite_all(A_desc,                   ! Write Aij into the BR-by-BC submatrix
     $           Aij,Aij_desc,i,j,BR,BC,request)           ! of A that starts at location (i,j).
            call mios_wait(request)                        ! Wait for the write to complete.
         end do
      end do

      call ooc_getrf(R,C,                                  ! Factor the R-by-C submatrix starting at
     $     Aij,1,1,A_desc,                                 ! (1,1) of the global out-of-core matrix.
     $     ipiv,info)                                      ! Aij is passed but not used.
c
c     After checking the error code, Fill b with a vector of unknowns and solve v = A^(-1)*v (omitted).
c
      call ooc_getrs('No transpose',R,1,                   ! Solve for a single right hand side.
     $     Aij,1,1,A_desc,                                 ! Out-of-core matrix containing the factors.
     $     ipiv,                                           ! Vector of pivot indices.
     $     v,1,1,v_desc,                                   ! The right hand side is overwritten by
     $     info)                                           ! the solution.
c
c     After checking the error code we can use the solution v (omitted).
c
      call mios_close(A_desc)                              ! Close the files.
      call mios_finalize()                                 ! Shut down the MIOS module.

      end program
```

Figure 2: A Fortran program that solves a general linear system out of core. The code shows the calls that create the required descriptors, store the matrix in blocks, factor the matrix out-of-core, and use the factors to solve a single linear system. The portions of the program in which submatrices are generated, the right hand side is initialized, and the solution vector is used are all omitted.

recursive algorithm starts the pipeline in every recursive call. Each BLAS uses only a single pipeline, even if it is implemented using multiple nested loops. Using a single pipeline complicates the index calculations required for prefetching, but it ensures that the pipeline startup cost is incurred only once per subroutine invocation.

Since LAPACK was designed specifically for computers with data caches, many its algorithms can be used in SOLAR and deliver acceptable performance. But an algorithm that works well on a computer with data cache can be slow when executed out-of-core, since the cost of a cache miss is typically much smaller than the cost of an I/O transfer. The level of data reuse required to achieve high performance in out-of-core computations is higher than the level of data reuse required to achieve high performance in data caches. Indeed, an experiment reported in Section 4 shows that an out-of-core implementation of LAPACK's right-looking LU decomposition algorithm can more than 3.75 times slower than an LU decomposition algorithm that is designed specifically for out-of-core execution. To achieve high-performance, we therefore modified LAPACK's algorithms or used other factorization algorithms.

The Cholesky factorization subroutine for positive-definite symmetric matrices uses a recursive algorithm, which is different from the iterative partitioned algorithms used by LAPACK and ScaLAPACK. An iterative algorithm would not allow pipelining either, because the iterative algorithm calls the BLAS, so pipelining could create read-before-write hazards. A recursive algorithm calls the BLAS on very large out-of-core matrices, which allows the BLAS flexibility in choosing block sizes.

We have implemented two algorithms for LU factorization with partial pivoting. One algorithm is a left-looking iterative algorithm and the other is a recursive algorithm. The left-looking algorithm is efficient when there is ample main memory, but the recursive algorithm is more efficient when main memory is small, as demonstrated by Table 4.

The basic left-looking algorithm uses two nested loops. In each iteration j of the outer loop a block of r columns is loaded from disk. In the each iteration $k < j$ of the inner loop a block of r columns which are all to the left of the current j block is loaded. The row exchanges that were necessary to pivot the k block are now applied to the j block so that their rows are stored using the same permutation, and then the k block is used to update the j block. The update consists of solving a triangular linear system with multiple right hand sides and of a matrix-matrix multiplication. When the updates from all the blocks to the left of block j have been applied to block j, it is factored. When the factorization is complete, each block is read again, the row exchanges that were generated after it was factored are applied, and it is written back. This algorithm, which was first proposed in [11], has two desirable features. First, the amount of I/O required for row exchanges is only one read and one write for every element in the strictly block-lower-triangular part of the matrix. Second, except for these writes, each element is written only once.

We have implemented a modified version of this algorithm, illustrated in Figure 3. The first modification, which was suggested to us by Bowen Alpern, is to use blocks with a different number of columns for the inner and outer loop. The number of times a matrix element is used before it is discarded is determined in this algorithm by the number of columns in j block. Increasing the width r_j of the j block by a factor of 2 reduces the number of elements read from disk by almost a factor of 2. To maximize r_j, we choose the width r_k of the k blocks to be as small as possible in order to conserve memory for the j blocks. Our implementation sets the number of columns r_k in the k block to be the number of columns in a primary block and allocates all the remaining memory to the j block. The second modification is to use pipelining on the k blocks. We do not use pipelining on the j blocks since this would reduce the size of r_j, because more in-core buffers are necessary. Since most of the I/O in this algorithm involves

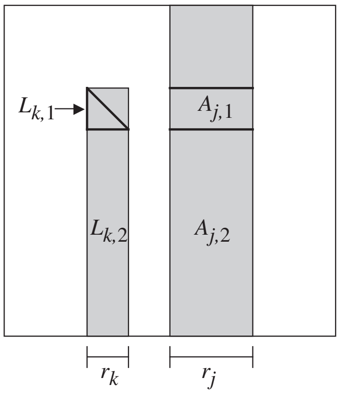

Figure 3: A schematic diagram of the left-looking out-of-core LU decomposition algorithm. The matrix is factored in blocks of r_j columns. Before a block is factored, it is updated by all the blocks of r_k columns to its left. In each update $A_{2,j}$ is replaced by $L_{2,k}^{-1}A_{2,j}$ and then $L_{3,k}A_{2,j}$ is subtracted from $A_{3,j}$.

reading k blocks and not j blocks, the decision not to pipeline the j blocks does not impact performance significantly.

The recursive algorithm works by factoring the left half of the matrix, applying the row exchanges from this factorization to the right half of the matrix, updating the right half using a large triangular solve and a large matrix-matrix multiply, factoring the updated right half, and finally applying row exchanges to the left half. Each half is factored using the same strategy. Toledo [24] recently proved that this algorithm uses only $\Theta(n^3/\sqrt{M})$ words worth of I/O in a simple two level memory hierarchy, where n is the order of the matrix and M is the size of main memory. In comparison, the left-looking algorithm uses $\Theta(n^4/M)$ words worth of I/O. Therefore, the recursive algorithm transfers a factor of $\Theta(n/\sqrt{M})$ fewer words between main memory and disk. Toledo also demonstrated that the recursive algorithm outperforms the right-looking algorithm that is used by LAPACK on several workstations because it generates fewer cache misses.

The main advantage of the recursive algorithm is that is uses blocked BLAS to update very large submatrices. The updates can achieve high performance with a constant amount of main memory that is not related to the size of matrix columns. The disadvantage of the recursive algorithm is that it perform row exchanges on all the columns in the matrix in every level of the recursion. Our implementation limits the cost of performing row exchanges by stopping the recurrence when the number of columns in the matrix to be factored is 1024 or less (this number is system-dependent). On matrices with at most 1024 columns the left-looking algorithm is used. Table 4 shows that this algorithm outperforms the left-looking algorithm when both are constrained to run in a small main memory.

We stress that the differences between the out-of-core algorithms in SOLAR and the algorithms in LAPACK and ScaLAPACK are only in the order that independent operations are performed, and therefore the numerical stability of SOLAR is equivalent to the stability of LAPACK.

Matrix Input-Output Subroutines

We currently have one main implementation of the matrix input-output module. It is used in both the parallel and the sequential version of SOLAR. It supports asynchronous I/O operations to matrices that are stored on one or more files or devices. It supports matrices stored in block-cyclic distributions on disks, in shared memory in-core buffers, and in block-cyclic distributions in distributed in-core buffers. There is only one temporary restriction on matrix layouts, which is that either the primary block size must divide the block size of the block-cyclic in-core data layout, or vice versa. (The condition applies to both the row dimension and the column dimension.) We plan to lift this restriction in the future. Our implementation can use a variety of low-level I/O interfaces, including low-level synchronous POSIX.1 system calls, low-level asynchronous POSIX.4 system calls, and direct asynchronous requests to an AIX device driver through a kernel extension (we use the term **raw I/O** to refer to this last interface). The POSIX system calls can be used to access files stored in PIOFS, IBM's parallel file system.

We illustrate how the MIOS module works by tracing an asynchronous read request and its associated wait request. A grid of processors reads a submatrix into a distributed in-core buffer, called the **application buffer**, in the following way:

1. Each processor decides which primary blocks of the submatrix it will read. The distribution of primary blocks to processors is a simple block distribution.

2. Each processor allocates a buffer to store all the primary blocks that it will read, determines the address of each one of them, and issues one low-level read request for each block. The buffer that stores the incoming data is called a **transfer buffer**. The address of a primary block is composed of the identity of the file or device that holds the block and the offset of the block within that file.

3. Each processor creates a **pending-request** structure and returns it to the caller. The pending-request structure identifies each pending low-level read request and specifies the data distribution of the application and transfer buffers.

4. When the application calls the wait subroutine in the MIOS and passes the pending-request structure, the MIOS waits for all low-level read requests to complete and then calls a data-redistribution subroutine.

5. The data-redistribution subroutine in the MIOS permutes the data in the transfer buffer into the requested layout in the application buffer.

The redistribution subroutine permutes a submatrix from a block layout with packed primary blocks into a block-cyclic layout with blocks arranged as subarrays in a Fortran array, or vice versa. The fact that primary blocks are packed, or contiguous in memory, implies that when the read or write request is for more than one row of primary blocks, data redistribution is necessary even on a uniprocessor. The subroutine permutes small submatrices rather than moving single elements in order to save on index calculation and in order to utilize optimized block copy subroutines in the BLAS. The size of these submatrices is usually either the primary block size or the block size of the block-cyclic data layout.

In the early development stages of SOLAR we have implemented another version of the MIOS module with support for debugging out-of-core algorithms that use asynchronous I/O. This version used synchronous I/O using calls to the stream-I/O subroutines in the standard C library. The feature that assists in debugging is the ability to perform reads and writes either when they are first requested or when the out-of-core algorithm waits for their completion. For example, by setting the appropriate flags so that reads are performed immediately but writes are performed as late as possible, it is possible to catch a read-before-write condition because the read returns incorrect data.

4 Experimental Results

This sections presents three sets of experimental results that substantiate our main claims. The first set of experiments was performed on a high-end workstation. These experiments show that SOLAR delivers high performance, and they therefore enable us to justify some of our design decisions. The second set of experiments was performed on a parallel computer. These experiments show that SOLAR can achieve high performance on parallel computers and that it can efficiently utilize multiple compute and I/O nodes. The third part of this section describes how we ported SOLAR to a laptop computer using public domain compilers and libraries. The success of this port supports our claim that SOLAR is indeed portable.

Experiments on a Workstation

The first set of experiments were performed on an IBM RS/6000 workstation with a 66.7 MHz POWER2 processor, 128 Kbytes 4-way set associative level-1 data-cache, a 1 Mbytes direct mapped level-2 cache, a 128-bit-wide main memory bus, and 256 Mbytes of main memory. The POWER2 processor is capable of issuing two double-precision multiply-add instructions per clock cycle. The operating system was AIX version 3.2.5 (AIX is IBM's version of UNIX). We used an IBM 2 Gbytes disk connected by a 16-bit channel to a 16-bit fast/wide SCSI-2 adapter. The disk was configured as one logical volume in one volume group, and was used only for storage of out-of-core matrices. (Volume groups and logical volumes are two disk abstractions that are used by AIX.) The logical volume was used by the program as a random access block device with no file system installed, in order to eliminate possible influences of the file system on the results. The SCSI-2 adapter was not used by any other devices during the experiments. We used the BLAS and Cholesky factorization subroutines from IBM's Engineering and Scientific Subroutine Library (ESSL). We used a non-shared POWER2-specific version of ESSL. We used the LU factorization and row exchange subroutines from LAPACK.

We chose to use the disk as a random-access block device rather than through a file system for two reasons. First, the I/O bandwidth achieved through a file system is often smaller than the bandwidth that can be achieved by performing I/O to a block device. Second, whereas the time it takes to perform an I/O operation in a file system is usually unpredictable due to caching and prefetching within the file system, in our experience the time it takes to perform I/O to a block device is usually quite consistent. In the context of this paper, using a file system would make the results of the experiments less repeatable and harder to analyze, so we chose not to use a file system. We also recommend that users attempt to use I/O to a block device whenever possible in order to improve performance. Our implementation of the MIOS allows several matrices to be stored on the same device by associating each out-of-core matrix with an absolute offset within the device.

Table 1 shows that the I/O bandwidth achieved by the raw I/O interface (I/O requests made directly to a device driver) can be more than twice the bandwidth achieved by low-level system calls. The rest of the workstation experiments therefore use raw I/O. Table 1 also demonstrates that there is essentially no difference in I/O performance between sequential I/O requests and requests that require a seek every 2 Mbytes. The bandwidth achieved by the asynchronous I/O low-level system calls degrades significantly when several operations are requested before any wait request is made. The slowdown probably occurs because several operating system processes, each one responsible for completing one I/O request, compete for the disk and cause poor scheduling of the disk-arm.

Table 2 documents SOLAR's performance on a typical out-of-core algorithm, the solution of a positive-definite symmetric linear system

	Factor					Solve (single right hand side)				
Matrix Order	Total Time	Compute Time	I/O Time	Blocks Read	Blocks written	Total Time	Compute Time	I/O Time	Blocks Read	Blocks written
2048	**34.8**	12.0	22.8 (0.90)	156	60	**10.9**	0.13	10.7 (0.32)	80	0
4096	**179**	95.5	83.4 (4.7)	944	224	**39.0**	0.55	38.6 (1.1)	288	0
6144	**511**	322	189 (13)	2844	476	**84.7**	1.26	83.0 (2.5)	624	0
8192	**1091**	763	328 (29)	6416	848	**147**	2.2	145 (4.3)	1088	0
10240	**2012**	1489	522 (53)	12076	1276	**228**	3.6	224 (6.6)	2496	0
12288	**3344**	2571	769 (89)	20448	1840	**327**	5.4	321 (9.5)	2400	0
14336	**5141**	4081	1055 (138)	31640	2476	**443**	6.9	435 (13)	3248	0

	Factor					Solve (single right hand side)				
Matrix Order	Total Time	Compute Time	I/O Time	Blocks Read	Blocks written	Total Time	Compute Time	I/O Time	Blocks Read	Blocks written
2048	**36.1**	11.8	24.3 (0.70)	112	64	**13.0**	0.18	12.8 (0.31)	96	0
4096	**168**	94.0	73.0 (3.5)	624	240	**43.0**	0.64	43.0 (1.2)	320	0
6144	**452**	319	133 (8.9)	1728	496	**90.0**	1.40	89.0 (2.6)	672	0
8192	**960**	757	202 (19)	3776	896	**155**	2.5	153 (4.5)	1152	0
10240	**1753**	1477	274 (32)	6832	1312	**239**	3.8	234 (6.9)	1760	0
12288	**2924**	2554	369 (53)	11376	1904	**337**	5.5	331 (9.8)	2496	0
14336	**4509**	4051	457 (79)	17504	2544	**453**	7.3	445 (13)	3360	0

Table 2: The performance characteristics of the out-of-core solution of a double-precision positive-definite symmetric linear system. The dimensions of the primary block are 256 by 256. The out-of-core linear algebra subroutines use blocks of dimensions 512 by 512 (upper table) or 1024 by 1024 (lower table). The program uses 16 Mbytes for in-core buffers when the block size is 512 and 64 Mbytes when the block size is 1024. That is, each block transfer moved 4 or 16 primary blocks. Times are reported in seconds and the numbers of blocks transferred is the number of primary blocks. The I/O time reported is the time spent in I/O which is not overlapped by computation. The times in parentheses are the redistribution times, which are also included in the overall I/O time.

	Factor (Left Looking)						Solve (single right hand side)				
Matrix Order	MB	Total Time	Compute Time	I/O Time	Blocks Read	Blocks written	Total Time	Compute Time	I/O Time	Blocks Read	Blocks written
2048	20	**56.1**	29.9	26.1 (1.1)	636	352	**13.1**	0.29	12.8 (0.43)	384	0
4096	40	**330**	219	111 (6.1)	3992	1472	**43.6**	0.89	42.7 (1.4)	1280	0
6144	60	**961**	702	259 (17)	12116	3360	**93.0**	1.90	81.0 (3.0)	2688	0
8192	80	**2119**	1646	472 (37)	27056	6016	**158**	3.5	155 (5.2)	4608	0
10240	100	**3897**	3140	754 (68)	50860	9440	**238**	5.0	233 (8.0)	7040	0
12288	120	**6521**	5408	1108 (112)	85576	13632	**346**	7.1	331 (11)	9984	0
14336	140	**10023**	8480	1537 (172)	133252	18592	**461**	9.6	451 (15)	13440	0

	Left Looking						Recursive					
Matrix Order	MB	Total Time	Comp. Time	I/O Time	Blocks Read	Blocks written	MB	Total Time	Comp. Time	I/O Time	Blocks Read	Blocks written
4096	12	**694**	215	478 (15)	12432	1520	16	**557**	210	356 (34)	9760	3392
6144	18	**2194**	695	1497 (48)	40280	3432	18	**1904**	685	1218 (300)	32176	9456
8192	24	**5041**	1629	3410 (110)	93472	6116	24	**3840**	1608	2230 (537)	68288	16896
10240	30	**9629**	3128	6497 (211)	180200	9560	30	**7241**	3092	4145 (1124)	131632	31160
12288	36	**16439**	5395	11039 (358)	308656	13776	36	**10577**	5320	5251 (1092)	205784	45024
14336	42	**25790**	8466	17317 (559)	487032	18760	42	**16783**	8404	8367 (2255)	326081	61432

Table 4: The performance characteristics of out-of-core algorithms for the LU factorization of a double-precision general matrix. The upper table shows the performance of the left-looking algorithm and of the solution of a linear system with a single right hand side using the factorization. The block sizes are $r_k = 128$ and $r_j = 512$ and pipelining is used on the j blocks. The lower table compares the performance of the left-looking and the recursive algorithms when both use a minimal amount of main memory. The left-looking algorithms use block sizes $r_k = r_j = 128$ and no pipelining. The recursive algorithm use a block size $r = 512$. The recursive algorithm switches to a left-looking algorithm with $r_k = r_j = 128$ and no pipelining when the number of columns in the matrix are 1024 or less. The dimensions of the primary block are 128 by 128. Times are reported in seconds and the numbers of blocked transferred is the number of primary blocks. The I/O time reported is the time spent in I/O which is not overlapped by computation. The times in parentheses are the redistribution times, which are also included in the overall I/O time.

Primary Block →	256 × 256		1024 × 1024	
I/O Interface ↓	Read	Write	Read	Write
POSIX.1 Synchronous	234	374	232	375
POSIX.4 Asynchronous	1210	857	233	375
AIX Raw	140	143	139	142

Table 1: The time in seconds to read and write a double-precision 8192-by-8192 matrix using three different low-level I/O interfaces and two choices of primary block dimensions. The total size of the matrix is $8 \times 8192 \times 8192 = 512$ Mbytes. The matrix is written and read in 1024-by-1024 blocks. When the order of the primary block is 1024, the matrix is read and written sequentially. When the order of the primary block is 256 the disk must seek once per 2 Mbytes transferred.

Block Order	MB MB	Total Time	Comp. Time	I/O Time	Blocks Read	Blocks Written
256	4	**1770**	766	995 (53)	11808	816
512	16	**1091**	763	328 (29)	6416	848
1024	64	**960**	757	202 (19)	3776	896

Table 3: The impact of increasing the block size on the main memory requirements and the running time of the out-of-core factorization of an 8192-by-8192 matrix. When the block size is 256, the amount of I/O is too large to be hidden by overlapping computation.

by factorization and triangular solution. The factorization algorithm is a recursive implementation of the Cholesky factorization. The most important fact that emerges from the table is that on large matrices SOLAR spends only 15–20% of the running time in I/O operations, even when the amount of main memory that is used is only 16 Mbytes. Quadrupling the amount of main memory that is used reduces the amount of I/O by almost one half. But since most of the I/O is already overlapped with computation, the effect on the running time is small. Table 3 shows that if the amount of memory is reduced to 4 Mbytes, the amount of I/O grows to the point where there is not enough computation to hide the I/O time.

Table 2 also shows that the solution time is small compared to the factorization time, even though almost all the solution time is spent in I/O operations.

The results reported in Table 2 show that the impact of unpipelined I/O in the recursive factorization and in the startup and shutdown of pipelines is small. The I/O bandwidth indicated in Table 1 is about 3.6 Mbytes/second. The total amount of I/O in the factorization of the matrix of order 14336 using blocks of order 1024 can therefore be performed in about 3000 seconds. More than 4000 seconds of computation give ample opportunity to hide the 3000 seconds of I/O. The results in the table show that 457 seconds were spent in I/O operations, including 79 seconds of data-redistribution time. Even if all of the $457 - 79 = 378$ seconds represent time spent in I/O operations that are not overlapped with computation due to the design of SOLAR (in the recursive factorization algorithm and in startups and shutdowns of pipelines in the out-of-core BLAS), that overhead still accounts for less than 10% of the overall running time.

Table 4 shows that SOLAR achieves high performance on a more difficult problem, the out-of-core LU factorization with partial pivoting of a general matrix. The algorithm that is used is the left-looking factorization algorithm described in Section 3. On the largest matrix almost 85% of the running time is spent in in-core computations. The amount of main memory that is required increases with the matrix order because the algorithm stores blocks of entire columns in memory in order to perform row exchanges efficiently.

Table 4 demonstrates that the recursively-partitioned algorithm outperforms the left-looking algorithm when both are tuned to use as little memory as possible. The amount of memory used by the left-looking algorithm can be reduced by not using pipelining and by choosing both block sizes r_j and r_k to be equal to the number of columns in a primary block, 128 in our matrices. (The size of the primary block, 128 Kbytes, was chosen to ensure high I/O bandwidth through the raw I/O interface.) The recursively-partitioned algorithm uses this variant of the left-looking algorithm when the number of columns in the matrix is 1024 or less, but uses a recursive strategy with calls to the BLAS to factor larger matrices. Since the amount of memory used by BLAS depends only on the block size in the algorithm but not on the order of the matrix, we choose a block size of 512 for the BLAS. This choice leads to a memory requirement of only 16 Mbytes, but allows most of the I/O to overlap computation. The table shows that the use of the BLAS in the recursively-partitioned improves performance without increasing the amount of memory required to factor large matrices. Specifically, the number of writes is larger and the number of reads is much smaller in the recursive algorithm. The total number of I/O operations is smaller in the recursive algorithm. The smaller amount of I/O together with the overlapping of some of the I/O with computation leads to smaller running times for the recursive algorithm.

To assess the usefulness of LAPACK's algorithms for out-of-core computation we translated LAPACK's right-looking LU decomposition algorithm into a SOLAR out-of-core subroutine. Factoring a 10240-by-10240 general matrix took 14730 seconds, out of which 3110 were spent in in-core computations and 11617 were spent in I/O (including 4843 for data redistributions). The factorization read 172080 primary blocks of size 128-by-128 and wrote 109760. The algorithm factored blocks of 512 columns and used 80 Mbytes of main memory. The factorization rate is 49 Mflops, which is acceptable given the small amount of effort we invested in the coding of the subroutine. The running time is large when compared to the 3897 seconds it took the left-looking algorithm to factor the same matrix (see Table 4). We conclude that whereas LAPACK's block-iterative algorithms can deliver acceptable performance, specialized out-of-core algorithms can run several times faster.

Experiments on Parallel Computers

The experiments were performed on two IBM SP2 parallel computers [1] which we denote by machine T and machine W. Machine T was configured with so-called thin nodes with 128 Mbytes of main memory as both the compute and I/O nodes and ran AIX version 4.1.3. Machine W was configured with wide-2 nodes having 512 Mbytes of main memory for both the compute and the I/O nodes, and ran AIX version 3.2.5. Thin nodes have a 66.7 MHz POWER2 processor, 64 Kbytes 4-way set associative level-1 data-cache, no level-2 cache, and a 64-bit-wide main memory bus. They also have smaller data-paths between the cache and the floating-point units than other POWER2-based SP2 nodes. Wide-2 nodes have a 77 MHz POWER2 processor, 256 Kbytes 4-way set associative level-1 data-cache, no level-2 cache, and a 256-bit-wide main memory bus. Thin nodes are the slowest SP2 nodes and wide-2 nodes are currently the fastest. We used IBM's parallel file system [7] (PIOFS) version 1.1. On machine T PIOFS used 4 I/O nodes. The parallel file system used 1 Gbytes on each node, allocated on a 2.2 Gbytes 16-bit SCSI disk. On machine W PIOFS used 4 I/O nodes. On each node, 1916 Mbytes were allocated for PIOFS on a 2.2 Gbytes 16-bit SCSI disk. On machine W the compute and I/O nodes are always distinct, but on machine T in some of the experiments some of the nodes may have served as both compute and I/O nodes. All the experiments use PIOFS through synchronous

b	Read	Write
256	20.2	19.9
128	20.7	20.3
64	21.4	21.0
32	22.6	22.3
16	25.7	25.7
8	32.8	33.2

Table 6: The time it takes, in seconds, to redistribute data when reading and writing a double-precision 8192-by-16384 matrix on Machine W as a function of the block size b in the block-cyclic data distribution. The primary block size is 256-by-256. The matrix was read and written in 1024-by-1024 blocks by 4 processors. The matrix was striped across 4 I/O nodes.

low-level POSIX.1 system calls. In all the experiments the message passing layer used the network interface in user-space mode and did not use interrupts. For in-core computations on distributed arrays we used IBM's Parallel Engineering and Scientific Subroutine Library (PESSL) version 1.1, which based on and completely compatible with ScaLAPACK [6], a public domain linear algebra package for linear algebra computations.[3] For in-core computations on arrays that are local to a processor we used IBM's Engineering and Scientific Subroutine Library (ESSL) version 2.2. We used POWER2-specific versions of all the libraries.

Table 5 shows the I/O bandwidths of the parallel file systems on the two machines. In all cases we used the default striping unit, 32 Kbytes, which means that every primary block was striped across all the I/O nodes. The I/O bandwidth improves when more I/O nodes are used, but the improvement is less than linear. For example, using 4 I/O nodes instead of 1 improves the read bandwidth by a factor of 3 on machine W. Using additional compute nodes improves the total bandwidth when there are several I/O nodes but can reduce the total bandwidth when there is only one I/O node. At least on the examples in the table, the differences in bandwidth due to the number of compute nodes are not large.

Caching of data on the I/O nodes is an important factor that affects performance in some of the experiments. The I/O bandwidths in some of the experiments reported in Table 5 is higher than the bandwidth that can be sustained on 4 disks, which is at most 4 Mbytes/second per disk. The effect of caching is much more pronounced on machine W, because its I/O nodes have larger main memories than the I/O nodes of machine T. We could not eliminate the effect of caching by the file system in our experiments because the amount of free disk space available to us was not large enough compared to the size of the main memories of the I/O nodes. We expect that in production environments the ratio of disk space to main memory on the I/O nodes will be much higher, and that caching by the file system will therefore not be able to significantly improve the performance of SOLAR. The improvement in performance due to caching shows, however, that the performance of PIOFS is limited by the performance of the disks and not by the performance of other components of the SP2 system.

Table 6 shows that the data redistribution subroutine is not sensitive to the granularity of block-cyclic distributions. Table 5 shows that the data redistribution times are higher when the block size is small because there are more index calculations when the block size is small and because more calls to the vector copy subroutine in the BLAS is made when the block size is small. The data-redistribution times are higher when the parallel version of SOLAR is used on a single

[3] PESSL also contains routines for Fourier transforms and related computations that are not part of ScaLAPACK.

Subroutine	$p = 1$	$p = 4$	$p = 16$
PDGEMM	213	163	143
PDSYRK	206	87	76
PDTRSM	206	46	21
PDPOTRF	195	66	48

Table 9: The performance in millions of floating-point operations per second per processor of 4 parallel in-core subroutines in PESSL. All the matrices are square with a 512-by-512 submatrix per processor. The processor grid is always square. PDGEMM is the general matrix multiply-add subroutine, PDSYRK is the symmetric matrix multiply-add subroutine, PDTRSM is the triangular solver, and PDPOTRF is the Cholesky factorization subroutine. The times were measured on an SP2 with thin nodes. The number of processors used is denoted by p. We used a 2-dimensional block layout for PDGEMM and PDSYRK and a 2-dimensional clock-cyclic layout with block size 64 for PDTRSM and PDPOTRF.

processor than when the sequential version is used. This is caused by copying the data three times within SOLAR in the parallel version versus copying the data only once in the sequential version. (The sequential version copies the data twice when the I/O is done directly from an application buffer without using a transfer buffer, but this case is not reported in the tables.)

Tables 7 and 8 show the performance characteristics of the out-of-core factorization of a positive-definite symmetric matrix. The data shows that using more I/O nodes improves performance and that using more compute nodes improves performance. Additional I/O nodes improve performance because they improve the total I/O bandwidth. Doubling the number of I/O nodes can improve the effective I/O bandwidth for the application, including data redistributions, by a factor of about 1.5. Additional compute nodes improve performance in two ways. First, they reduce the time for in-core computations. The speedup is by more than a factor of 2 when 4 nodes are used instead of 1. The reason that the speedup is not closer to 4 is mainly the poor performance of the triangular solve subroutine in PESSL that does not scale well. Second, additional compute nodes increase the size of the available main memory and they therefore allow us to use larger block sizes in the algorithms. Since the number of I/O's is almost halved when the order of the blocks doubles, using 4 processors instead of 1 almost halves the total amount of I/O the factorization requires. This too, of course, improves performance.

Table 9 shows that some of the in-core matrix subroutines in PESSL, especially the triangular solver, do not scale well. The four subroutines whose performance is reported are the ones that are used by the out-of-core Cholesky factorization. Other experiments, not reported here, show that the low performance is not caused by the power-of-2 order of the matrices. The performance of the triangular solver improves when the message passing layer of the SP2 uses interrupts, but this causes the overall performance of the out-of-core factorization to degrade because other operations slow down. While this performance problem has nothing to do with the design of SOLAR, it impacts the performance that SOLAR delivers because SOLAR uses these subroutines. The impact decreases as the problem size increases, because the fraction of matrix multiplications within the total operation counts increases when the problem sizes increases. For example, the in-core computation time on Machine W in the factorization of a matrix of order 8192 is 147 Mflops/processor, but the rate increases to 175 Mflops/processor when the order of the matrix increases to 14336.

		Machine W (Wide-2 Nodes)						Machine T (Thin Nodes)			
# I/O Nodes →		1		2		4		2		4	
IC Buffer ↓	p	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write
Shared	1	186 (4.1)	168 (4.2)	108 (4.1)	89 (4.2)	66 (4.1)	67 (4.2)	259 (15)	359 (15)	192 (16)	241 (16)
Distributed	1	181 (12)	168 (12)	105 (12)	90 (12)	72 (12)	74 (12)	278 (43)	410 (43)	217 (44)	287 (43)
Distributed	2	202 (18)	182 (18)	117 (18)	99 (18)	68 (18)	70 (18)	276 (40)	315 (38)	194 (38)	215 (38)
Distributed	4	209 (20)	184 (20)	124 (20)	101 (20)	51 (21)	64 (20)	268 (31)	360 (31)	188 (32)	223 (30)

Table 5: The time it takes, in seconds, to read and write a double-precision 8192-by-16384 matrix on two SP2 computers. The matrix is written and read in 1024-by-1024 blocks. Distributed buffers have a 2-dimensional block distribution. The dimensions of the primary block are 256 by 256. The table shows how the I/O time depends on the number of I/O nodes and on the number p of compute nodes. When only one compute node is used, the table shows the time to read and write the matrix using both a shared memory buffer and a distributed memory buffer. Even though the two cases produce exactly the same result, the redistribution code that is used is different. There was not enough space in the file system on Machine T to store the matrix on a single I/O node. The times in parentheses are the redistribution times, which are also included in the overall I/O time.

	1 Processor					4 Processors				
Matrix Order	Total Time	Comp. Time	I/O Time	Blocks Read	Blocks written	Total Time	Comp. Time	I/O Time	Blocks Read	Blocks written
2 I/O Nodes										
8192	**1823**	866	953	6416	848	**956**	460	479	944	244
10240	**3433**	1695	1729	12076	1276	**1723**	833	875	1708	328
12288	**5818**	2923	2882	20448	1840	**2838**	1364	1453	2844	476
4 I/O Nodes										
8192	**1596**	866	726	6416	848	**808**	455	338	944	224
10240	**3023**	1690	1323	12076	1276	**1430**	831	579	1708	328
12288	**5132**	2919	2198	20448	1840	**2340**	1364	957	2844	476
14336	**8045**	4633	3389	31964	2476	**3560**	2083	1452	4376	636

Table 7: The performance characteristics of the out-of-core Cholesky factorization on an SP2 computer with **thin nodes**. The matrix was stored on 2 or 4 nodes of the parallel file system. The block size in the algorithm was 512-by-512 per processor. The dimensions of the primary block are 256 by 256. There was not enough disk space to factor a matrix of order 14336 on 2 I/O nodes. The number of primary blocks read and written is per processor. The times in parentheses are the redistribution times, which are also included in the overall I/O time.

	1 Processor					4 Processors				
Matrix Order	Total Time	Comp. Time	I/O Time	Blocks Read	Blocks written	Total Time	Comp. Time	I/O Time	Blocks Read	Blocks written
4 I/O Nodes										
8192	**813**	630	183	3776	896	**381**	311	69	624	240
10240	**1551**	1230	321	6832	1312	**670**	565	105	1056	336
12288	**2650**	2124	525	11376	1904	**1098**	924	174	1728	496
14336	**4161**	3373	788	17504	2544	**1687**	1407	240	2592	656

Table 8: The performance characteristics of the out-of-core Cholesky factorization on an SP2 computer with **wide-2 nodes**. The matrix was stored on 2 or 4 nodes of the parallel file system. The block size in the algorithm was 1024-by-1024 per processor. Block size in block cyclic is 64 for trsm, syrk, block for gemm. The dimensions of the primary block are 256 by 256. The number of primary blocks read and written is per processor. The times in parentheses are the redistribution times, which are also included in the overall I/O time.

Porting Experience

We ported SOLAR to a laptop computer to demonstrate its portability. Specifically, we ported the system, which was initially developed on an IBM RS/6000 workstation running AIX to an IBM ThinkPad 755CX running OS/2 version 3.0. The target machine had a 75 MHz Intel Pentium processor with a 256 Kbytes level-2 cache, 24 Mbytes of main memory, and an 810 Mbytes disk. We used only public domain software and tools on this system in order to demonstrate that SOLAR can be ported without any proprietary tools. We used the EMX 0.9a port of GCC 2.6.3 (the GNU C compiler), the F2C Fortran to C translator, and CBLAS and CLAPACK, which are C versions of LAPACK and the BLAS that are based on a translation by F2C. All of these tools are available from online archives of public domain software. Downloading and installing these packages on the ThinkPad took a few hours.

One person ported SOLAR in the course of one day. At the end of the day, we were able to factor a general double-precision matrix of order 2048, whose total size is 32 Mbytes using 8 Mbytes of main memory. In this environment, the MIOS module uses the low-level synchronous POSIX.1 I/O interface. We had to make three small changes in SOLAR, one of which was anticipated, in order to get it to compile and run. The anticipated change was the replacement of an RS/6000-specific timing routine by a more generic routine based on the System V `ftime` system call. This timing routine is used by SOLAR for profiling only, and both versions are less than 25-lines long. The second change was necessary because F2C adds an underscore to the names of all subroutines. We therefore had to create include files with macros that add an underscore to the names of subroutines in LAPACK and the BLAS. We also had to remove the underscores from the translated version of our test program, which is written in Fortran. These underscores were removed using a single search and replace operation with a regular expression in EMACS. Following these two changes SOLAR compiled and passed some of the tests. It failed to pass the rest of the tests since file read operations returned too few bytes. This caused SOLAR to abort with an appropriate error message. The problem was caused by the fact that in this version of GCC, the `open` system call should be called with a nonstandard flag that indicates that the file contains binary rather than text data. Adding the flag solved the problem and enables SOLAR to pass all our test.

The fact that we were able to quickly port SOLAR with few difficulties to an environment with a different operating system, no Fortran compiler, and using only public domain tools indicates that SOLAR is indeed portable. The port enabled us to solve problems that cannot be solved in-core on this system. Portability does not come at the cost of performance. The performance of the out-of-core solver was very good compared to the performance of in-core solvers. The first system that we solved, whose order was 2048, took 4072 seconds to factor and 53 seconds to solve for a single in-core right hand side. The out-of-core factorization rate, 1.41 Mflops, is more than 85% of the in-core factorization rate, which is about 1.65 Mflops. (The utilization rate is high due to the design of SOLAR, but also due to a the high ratio of I/O bandwidth to computation rate. The I/O bandwidth of this machine is high, about 1 Mbytes per second, and the computation rate is relatively low due to a processor with mediocre floating-point performance and nonoptimized BLAS.)

5 Conclusions

This paper describes SOLAR, a portable library for high-performance out-of-core dense matrix computations. SOLAR is designed to be integrated into ScaLAPACK and to eventually provide much of the functionality of LAPACK and ScaLAPACK for matrices stored on disks.

We believe that the overall organization of SOLAR, the design of its interfaces, and the design of key subroutines address the three main goals of the library: portability, high-performance, and providing a wide range of solvers. The overall organization helps to achieve these goals by utilizing existing in-core dense linear algebra libraries and by enabling us to reuse LAPACK's algorithms. The design of the interfaces helps to achieve the goals by simplifying and modularizing SOLAR. The interfaces do so in ways that do not significantly impact the library's performance. Key subroutines in SOLAR, in particular the out-of-core subroutines and the matrix I/O subroutines, are designed to efficiently utilize both I/O bandwidth and computational resources.

SOLAR works on distributed-memory parallel computers, workstations, and personal computers. The experiments that are reported in the paper show that SOLAR achieves high performance in all of these environments, in the sense that most of its running time is spent in in-core computations rather than in I/O operations.

The portability of the library and the use of a high-level matrix input-output interface makes the library a convenient platform for research on out-of-core algorithms, I/O optimization, and parallel I/O algorithms.

Acknowledgments

Thanks to Jack Dongarra for his suggestions regarding the design of the library and for his support for incorporating the library into ScaLAPACK. Thanks to Ramesh Agarwal, Bowen Alpern, and Rob Schreiber for helpful discussions. Steve Watts of the IBM Santa Teresa Lab wrote the kernel extension that we used for raw I/O in AIX. Thanks to the anonymous referees for several helpful suggestions.

References

[1] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, PA, 2nd edition, 1994. Also available online from `http://www.netlib.org`.

[3] D. W. Barron and H. P. F. Swinnerton-Dyerm. Solution of simultaneous linear equations using a magnetic tape store. *Computer Journal*, 3:28–33, 1960.

[4] Jean-Philippe Brunet, Palle Pedersen, and S. Lennart Johnsson. Load-balanced LU and QR factor and solve routines for scalable processors with scalable I/O. In *Proceedings of the 17th IMACS World Congress*, Atlanta, Georgia, July 1994. Also available as Harvard University Computer Science Technical Report TR-20-94.

[5] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. Technical Report CS-95-292, University of Tennessee, May 1995.

[6] J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A scalable linear algebra for distributed memory concurrent computers. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127, 1992. Also available as University of Tennessee Technical Report CS-92-181.

[7] P. F. Corbett, D. G. Feitelson, J.-P. Prost, G. S. Almasi, S. J. Baylor, A. S. Bolmarcich, Y. Shu, J. Satran, M. Snir, R. Colao, B. D. Herr, J. Kavaky, T. R. Morgan, and A. Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 34(2):222–248, 1995.

[8] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. Technical Report PCS-TR93-188, Dept. of Math and Computer Science, Dartmouth College, March 1993. Revised 9/20/94. An ealier version appeared in the Proceedings of the 1993 DAGS/PC Symposium.

[9] Jack J. Dongarra, Jeremy Du Cruz, Sven Hammarling, and Ian Duff. An set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[10] Jack J. Dongarra, Jeremy Du Cruz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[11] J. J. Du Cruz, S. M. Nugent, J. K. Reid, and D. B. Taylor. Solving large full sets of linear equations in a paged virtual store. *ACM Transactions on Mathematical Software*, 7(4):527–536, 1981.

[12] Charbel Farhat. Large out-of-core calculation runs on the IBM SP2. *NAS News*, 2(11), August 1995.

[13] Nikolaus Geers and Roland Klees. Out-of-core solver for large dense nonsymmetric linear systems. *Manuscripta Geodetica*, 18(6):331–342, 1993.

[14] Roger G. Grimes and Horst D. Simon. Solution of large, dense symmetric generalized eigenvalue problems using secondary storage. *ACM Transactions on Mathematical Software*, 14(3):241–256, 1988.

[15] Kenneth Klimkowski and Robert van de Geijn. Anatomy of an out-of-core dense linear solver. In *Proceedings of the 1995 International Conference on Parallel Processing*, 1995. To appear.

[16] David Kotz. Disk-directed I/O for an out-of-core computation. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, pages 159–166, August 1995. Also available as Dartmouth College Technical Report PCS-TR95-251.

[17] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprogram for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[18] David A. Patterson and John L. Hennessy. *Computer Architecture A Quantitative Approach, Second Edition*. Morgan Kaufmann, San Francisco, 1995.

[19] Hans Riesel. A note on large linear systems. *Mathematical Tables and Other Aids to Computation*, 10:226–227, 1956.

[20] J. Rutledge and H. Rubinstein. High order matrix computation on the UNIVAC. Presented at the meeting of the Association for Computing Machinery, May 1952.

[21] Joseph Rutledge and Harvey Rubinstein. Matrix algebra programs for the UNIVAC. Presented at the Wayne Conference on Automatic Computing Machinery and Applications, March 1951.

[22] M. M. Stabrowski. A block equation solver for large unsymmetric linear equation systems with dense coefficient matrices. *International Journal for Numerical Methods in Engineering*, 24:289–300, 1982.

[23] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. Technical Report CACR-103, Scalable I/O Initiative, Center for Advanced Computing Research, Caltech, June 1995. Submitted to a special issue of *Scientific Programming* on implementations of HPF.

[24] Sivan Toledo. Locality of reference in LU decomposition with partial pivoting. Technical Report RC20344, IBM T.J. Watson Research Center, Yorktown Heights, NY, January 1996.

[25] David Womble, David Greenberg, Stephen Wheat, and Rolf Riesen. Beyond core: Making parallel computer I/O practical. In *Proceeings of the 1993 DAGS/PC Symposium*, pages 56–63, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies. Also available online from `http://www.cs.sandia.gov/˜dewombl`.