# Progress Towards Optimizing the PETSc Numerical Toolkit on the Cray X1 [*]

Richard Tran Mills, Eduardo F. D'Azevedo, and Mark R. Fahey
Oak Ridge National Laboratory
Oak Ridge, TN 37831, USA.

## Abstract

Modern iterative solver packages have targeted superscalar computer architectures, and typically exhibit poor out-of-box performance on vector processor machines. In this work, we describe initial work on algorithmic modifications to the popular PETSc scientific toolkit to improve the vectorizability of its iterative linear system solvers. We describe our implementation of a PETSc matrix class that uses a simple vectorizable algorithm to perform sparse matrix-vector multiplication with compressed sparse row (CSR) format matrices. Performance tests using our kernel in codes that solve a variety of physics problems on the Cray X1 indicate that speedups of an order of magnitude or better for matrix-vector multiplication are typical. Further work remains, however, as applications are still slowed by poor vectorization of preconditioning operations.

## 1 Introduction

A great many systems of interest to scientists and engineers are modeled by analytically intractable boundary value problems. Scientific application codes often approach such problems by considering a discretized form of the governing partial differential equations (PDEs). In such codes, the bulk of the computation usually lies in the solution of a linear system of equations $Ax = b$. The matrix $A$ is usually sparse, consisting of many zero elements. In practical applications, $A$ also tends to be large, as increasing the resolution of a simulation rapidly increases the dimension of the matrix. One way to solve such linear systems is through so-called direct methods, which are essentially variations on Gaussian elimination. Direct methods are general and robust, but it can be difficult to limit fill-in (increase in the number of nonzero elements) when using them, and it can be extremely challenging to get good scalability when solving huge matrix problems on many parallel processors. Consequently, iterative methods are popularly employed to solve such systems. These methods are not guaranteed to always converge to a solution, but they are comparatively easy to parallelize and usually require less time than a direct method when a solution can be found.

The most widely used iterative methods are Krylov subspace methods, which build a vector space from which increasingly accurate approximate solutions are extracted. Although understanding the theory behind Krylov methods is non-trivial, these methods are relatively easy to implement, as they involve only matrix-vector products, dot products, and vector updates. Krylov methods are rarely employed without some form of preconditioning to improve convergence. Preconditioning can be viewed as transforming a linear system $Ax = b$ into an equivalent system $M^{-1}Ax = M^{-1}b$ that has the same but more easily found solution; obviously, it is desirable that $M^{-1}$ approximate $A^{-1}$ in some fashion. In the absence of a preconditioner based on some specific knowledge about the physics or other attributes of a given problem, some form of incomplete LU factorization (ILU) is usually employed as a generic preconditioner.

Many scientific simulation codes spend the majority of their time applying Krylov methods, specifically in the matrix-vector multiply required at each iteration and the sparse triangular solve required to apply the ILU preconditioner. Unfortunately, we have found that most modern software packages for iterative solution of linear systems were designed with

[*]This research was sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL), and used resources of the Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The submitted manuscript has been authored by contractors of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

scalar computers in mind, and hence perform poorly on vector architectures such as the Cray X1. In this paper we present some initial results from ongoing work on algorithmic changes to improve the vectorizability of kernels within the popular PETSc toolkit. PETSc (the Portable, Extensible Toolkit for Scientific Computation) is a suite of data structures and associated routines for the scalable solution of problems arising from systems modeled by partial differential equations [1]. It employs an object-oriented design that shields application programmers from underlying, complicated data-structures and message passing by providing abstract objects. We will demonstrate how we have integrated alternative data structures and kernels for sparse matrix-vector kernels into the PETSc object-oriented framework to improve performance of applications that rely on PETSc iterative solvers on the Cray X1.

## 2 Background: sparse matrix-vector multiplication

Our initial efforts have focused on improving the performance of sparse matrix-vector multiplications ("mat-vecs") on the Cray-X1, in which nearly all iterative solvers spend a great deal of their time. Here we present background on some common sparse matrix storage formats and the algorithms used to compute matrix-vector products with them. Readers desiring further information can refer to [2, 9].

The most widely-used general format for storing sparse matrices is the compressed sparse row (CSR) format, also known as compressed row storage (CRS), Yale sparse matrix, or AIJ format. The format is very space-efficient, storing only nonzero entries. In CSR, the matrix is stored in three arrays, which we denote `val`, `col_ind`, and `row_ptr`. The array `val` stores the nonzero elements in row by row fashion, from row 1 to $n$. The integer array `col_ind` contains the column index of each entry in `val`, i.e., if `val`(k) $= a_{ij}$ then `col_ind`(k) $= j$. The integer array `row_ptr` is used to track the beginning of each row of the matrix in the arrays `val` and `col_ind`, i.e., the `row_ptr`(i) is the position in `val` and `col_ind` where the ith row begins. For example, the matrix

$$A = \begin{pmatrix} 11 & 0 & 0 & 14 & 0 \\ 21 & 22 & 0 & 24 & 0 \\ 31 & 0 & 33 & 34 & 35 \\ 0 & 0 & 43 & 44 & 0 \\ 0 & 0 & 0 & 0 & 55 \end{pmatrix} \tag{1}$$

is represented in CSR format in Figure 1.

Matrix-vector multiplication for CSR format is fairly straightforward. The multiplication proceeds in row by row fashion, moving through the `val` and `col_ind` values with unit stride. Figure 2 describes the algorithm. The algorithm performs acceptably on scalar computer architectures, but tends to perform poorly on vector machines. This is because vectorization across the row index `J` is limited by the number of nonzeros (`IEND` $-$ `ISTART` $+ 1$) per row, and with most discretization schemes, this number is usually only a fraction of the vector register length. For example, using a "star"-type finite difference stencil in three dimensions results in only seven nonzeros per row.

If the number of nonzeros per row is fairly constant, the ELLPACK-ITPACK format (ELL) [6] can yield much better performance than CSR on vector machines. ELL stores the matrix in two rectangular arrays, `val` and `col_ind`, of dimension `N` $\times$ `NZMAX`, where `N` is the number of rows and `NZMAX` is the maximum number of nonzero elements per row. To construct the ELL data structure, all nonzero elements of the matrix are shifted left, and then columns of the shifted "matrix" are stored consecutively in `val`. Rows that have fewer than `NZMAX` nonzeros will result in padding with zeros on the right to give all rows in `val` equal length. Element $(i, j)$ of `col_ind` stores the column number of element $(i, j)$ of `val`. Note that, for entries in `col_ind` that correspond to zero elements, the "column number" is chosen to equal the row number. This is somewhat arbitary, because when forming a matrix-vector product, the corresponding element from the vector will be multiplied by zero. Figure 2 illustrates the ELL storage scheme for the matrix in Equation 1.

The advantage of ELL format is that matrix multiplication is easily vectorized with a vector length essentially equal to the number of rows `N`. This algorithm is presented in Figure 4. A drawback of this algorithm is that it generates a significant amount of memory traffic because the complete vector `Y` will be repeatedly read in and written out again. A simple variant of this algorithm that can decrease memory traffic is shown in Figure 5. This algorithm employs a "strip-mining" approach, working with an array `YP` of small enough size to fit into vector registers, thus avoiding repeatedly moving `Y` to and from memory.

The ELL format enables good mat-vec performance on vector machines, but it has limited applicability: it is highly wasteful in terms of both storage and computation if the number of nonzeros differs widely per row. In the worst-case scenario, a matrix in which one row is full but all others are very sparse will require two full `N`$\times$`N` arrays consisting al-

| val | 11 | 14; | 21 | 22 | 24; | 31 | 33 | 34 | 35; | 43 | 44; | 55 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| col_ind | 1 | 4; | 1 | 2 | 4; | 1 | 3 | 4 | 5; | 3 | 4; | 5 |

| row_ptr | 1 | 3 | 6 | 10 | 12 | 13 |
|---|---|---|---|---|---|---|

Figure 1: The matrix (1) stored in CSR format.

```
DO I=1,N
  ISTART = ROW_PTR(I);IEND = ROW_PTR(I+1)-1
  YI = 0.0
  DO J=ISTART,IEND
     YI = YI + VAL(J) * X( COL_IND(J) )
  ENDDO
  Y(I) = YI
ENDDO
```

Figure 2: Matrix multiply in CSR format

| | | | | |
|---|---|---|---|---|
| val(:,1) | 11 | 14 | 0 | 0 |
| val(:,2) | 21 | 22 | 24 | 0 |
| val(:,3) | 31 | 33 | 34 | 35 |
| val(:,4) | 43 | 44 | 0 | 0 |
| val(:,5) | 55 | 0 | 0 | 0 |

| | | | | |
|---|---|---|---|---|
| col_ind(:,1) | 1 | 4 | 1 | 1 |
| col_ind(:,2) | 1 | 2 | 4 | 2 |
| col_ind(:,3) | 1 | 3 | 4 | 5 |
| col_ind(:,4) | 3 | 4 | 4 | 4 |
| col_ind(:,5) | 5 | 5 | 5 | 5 |

Figure 3: The ELLPACK-ITPACK (ELL) representation of the matrix from Equation 1.

```
Y(1:N) = 0.0
DO J=1,NZMAX
  Y(1:N) = Y(1:N) +  VAL(1:N,J)*X( COL_IND(1:N,J) )
ENDDO
```

Figure 4: Matrix multiply in ELLPACK format

```
DO I=1,N,NB
   IEND = MIN(N,I+NB-1)
   M = IEND-I+1
   YP(1:M) = 0.0
!  ---------------------------------------------
!  Consider YP(1:M) as vector registers
!  NB is multiple of the size of vector registers
!  ---------------------------------------------
   DO J=1,NZ
      YP(1:M) = YP(1:M) + VAL(I:IEND,J) * X( COL_IND(I:IEND,J) )
   ENDDO
   Y(I:IEND) = YP(1:M)
ENDDO
```

Figure 5: “Strip-mined” variant of matrix multiply in ELLPACK format

most entirely of zeros! The jagged diagonal (JAD) format is a generalization of the ELL format that addresses this shortcoming. Construction of the JAD data structure begins by determining a permutation vector `perm` that orders the rows of the matrix by decreasing number of nonzero elements. A number of "jagged diagonals" are then constructed from the permuted matrix $PA$ as follows: The first jagged diagonal consists of the leftmost element of the first row of $PA$, followed by the leftmost element of the next row, and so on for each row. The second jagged diagonal is constructed similarly from the next to leftmost element of each row of $PA$, and so on. The jagged diagonals are then stored consecutively in an array `jdiag`. An integer array `col_ind` of the same dimension stores the column index for each element of `jdiag`, and an array `jd_ptr` denotes where each jagged diagonal begins in `jdiag`. Figure 6 depicts the JAD data structure for the example matrix 1. Long vector lengths for matrix-vector multiplication are easily obtained with JAD by vectorizing down along the rows (see the algorithm in Figure 7). One drawback of JAD is that construction of the jagged diagonals often starts with the CSR data structure, so significant extra storage is needed to copy the matrix into the JAD structure.

## 3 The CSRP matrix-vector multiplication algorithm

The vectorizable sparse matrix-vector multiplication algorithm that we have used to speed up PETSc is the CSRP (CSR with Permutation) algorithm originally described in [4]. It works with a matrix stored in CSR format, and computes an additional permutation vector that is used to group together rows with the same number of nonzeros. The matrix-vector product is computed one group at a time, with the computation for each group being carried out in a manner similar to the ELLPACK algorithm. Unlike in the ELLPACK algorithm, however, the data remain in place and are accessed indirectly using the permutation vector. Although this involves non-unit stride access to the data, the CSRP algorithm nevertheless yields considerably better performance than the standard CSR multiply because of much better vector lengths. Note that if the extra storage can be spared, the groups can be copied into ELLPACK format to allow unit-stride access to the `val` and `col_ind` arrays, improving performance. Figure 8 depicts the CSRP algorithm.

## 4 A CSRPERM matrix class for PETSc

PETSc is written in C, but utilizes a fully object-oriented programming model, employing its own function tables and dispatch mechanism. All PETSc objects are derived from an abstract base object, such as the `Mat` (matrix) object. The `Mat` object has a variety of instantiations, corresponding to different storage formats. The most widely-used instantiation is the AIJ (compressed sparse row storage) matrix type, which is actually implemented using two matrix classes, `MATSEQAIJ` and `MATMPIAIJ`. `MATSEQAIJ` stores a matrix residing on a single processor, while `MATMPIAIJ` stores a matrix across several processors (as a collection of `MATSEQAIJ` matrices that hold the local portions of each matrix). We have seamlessly integrated support for our CSRP algorithm into PETSc by creating a `CSRPERM` matrix type consisting of two classes, `MATSEQCSRPERM` and `MATMPICSRPERM` that inherit most of the attributes and methods of `MATSEQAIJ` and `MATMPIAIJ`, respectively. Our `CSRPERM` classes need only override a select few methods supported by `AIJ`.

In PETSc, a `Mat` object `A` is built into a particular type by calling `MatSetType(Mat mat, MatType matype)`. If the matrix type is `MATSEQCSRPERM`, then PETSc will call our internal `MatCreate_SeqCSRPERM` routine, shown in Figure 9. One can see that all this routine does is build a matrix of type `MATSEQAIJ`, then call our internal routine that converts a `MATSEQAIJ` matrix into a `MATSEQCSRPERM` one. Figure 10 displays some fragments of the conversion routine. In lines 7–8, we create a `Mat_SeqCSRPERM` data structure, which will store all of the additional data needed by the CSRP algorithm, and stash its address in the '`spptr`' field, which is provided in the generic `Mat` type specifically as a means of pointing to extra data needed by new matrix implementations. In lines 12–16 we set the function pointers for the AIJ methods we need to override. We must override `duplicate` and `destroy` because those methods must know about the extra data required by the CSRP algorithm. We override `mult` and `multadd` to replace those kernels with our CSRP mat-vec, and we override `assemblyend` because that is where the calculation of the permutation vector (and, optionally, arrangement of data into ELL blocks) must occur.

A call to `MatCreate_SeqCSRPERM` (and the subsequent call to the conversion routine), creates an empty `MATSEQCSRPERM` object but does not do any setup of the actual matrix data structure. In order to allow overlap of communication and computation, actual matrix assembly in PETSc occurs

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| jdiag | 31 | 21 | 11 | 43 | 55; | 33 | 22 | 14 | 44; | 34 | 24; | 35 |
| col_ind | 1 | 1 | 1 | 3 | 5; | 3 | 2 | 4 | 4; | 4 | 4; | 5 |

| | | | | |
|---|---|---|---|---|
| jd_ptr | 1 | 6 | 10 | 12 |

| | | | | | |
|---|---|---|---|---|---|
| perm | 3 | 2 | 1 | 4 | 5 |

Figure 6: The matrix (1) stored in JAD format.

```
Y(1:N) = 0.0
DO J=1,NUM_JD
    K1 = JD_PTR(J)
    K2 = JD_PTR(J+1) - 1
    LEN = JD_PTR(J+1) - K1
    Y(1:LEN) = Y(1:LEN) + JDIAG(K1:K2) * X( COL_IND(K1:K2) )
ENDDO
```

Figure 7: Matrix multiply in JAD format

```
DO IGROUP=1,NGROUP
  JSTART = XGROUP(IGROUP)
  JEND = XGROUP(IGROUP+1)-1
  NZ = NZGROUP(IGROUP)
! ------------------------------------------------------------
! Rows( IPERM(JSTART:JEND) ) all have same NZ nonzeros per row
! ------------------------------------------------------------
  DO I=JSTART,JEND,NB
    IEND = MIN(JEND,I+NB-1)
    M = IEND - I + 1
    IP(1:M) = ROW_PTR( IPERM(I:IEND) )
    YP(1:M) = 0.0
!   ----------------------------------------
!   Consider YP(:), IP(:) as vector registers
!   ----------------------------------------
    DO J=1,NZ
       YP(1:M) = YP(1:M) + VAL( IP(1:M) ) * X( COL_IND(IP(1:M)) )
       IP(1:M) = IP(1:M) + 1
    ENDDO
  ENDDO
  Y( IPERM(I:IEND) ) = YP(1:M)
ENDDO
```

Figure 8: Matrix multiply for CSR with permutation. IPERM is the permutation vector, and XGROUP points to the beginning indices of groups in IPERM.

```
PetscErrorCode MatCreate_SeqCSRPERM(Mat A)
{
  PetscObjectChangeTypeName((PetscObject)A,MATSEQCSRPERM);
  MatSetType(A,MATSEQAIJ);
  MatConvert_SeqAIJ_SeqCSRPERM(A,MATSEQCSRPERM,MAT_REUSE_MATRIX,&A);
  return(0);
}
```

Figure 9: The constructor routine that is registered with PETSc for the MATSEQCSRPERM class.

```
PetscErrorCode MatConvert_SeqAIJ_SeqCSRPERM(Mat A,MatType type,
   MatReuse reuse,Mat *newmat)
{
  Mat            B = *newmat;
  Mat_SeqCSRPERM *csrperm;
...
  ierr = PetscNew(Mat_SeqCSRPERM,&csrperm);CHKERRQ(ierr);
  B->spptr = (void *) csrperm;
...
  /* Set function pointers for methods that we inherit from AIJ but
   * override. */
  B->ops->duplicate   = MatDuplicate_SeqCSRPERM;
  B->ops->assemblyend = MatAssemblyEnd_SeqCSRPERM;
  B->ops->destroy     = MatDestroy_SeqCSRPERM;
  B->ops->mult        = MatMult_SeqCSRPERM;
  B->ops->multadd     = MatMultAdd_SeqCSRPERM;
...
  ierr = PetscObjectChangeTypeName((PetscObject)B,MATSEQCSRPERM);CHKERRQ(ierr);
  *newmat = B;
  PetscFunctionReturn(0);
}
```

Figure 10: Portions of the internal routine for converting a `MATSEQAIJ` into a `MATSEQCSRPERM`.

in two steps: invoking the `assemblybegin` method starts the process, and a matching `assemblyend` call finalizes the process. Because creating the CSR-PERM information proceeds from an AIJ (CSR) data structure, the `MATSEQCSRPERM` assembly can use the `MATSEQAIJ` routines to first construct an AIJ matrix and then proceed from there. The `assemblybegin` method for `MATSEQCSRPERM` is identical to that for `MATSEQAIJ`. The `assemblyend` method, depicted in Figure 11, simply calls the `assemblyend` method for the `MATSEQAIJ` class (a pointer to which has been stashed by the `MatConvert` call), and then constructs the permutation for `MATSEQCSRPERM` using a bucket sort. (Note that the `inode.use` attribute is set to false before the `MATSEQAIJ` assembly; this ensures that the AIJ "inode" matrix-vector multiplication routines, which take advantage of rows with identical nonzero structure, will not be preferred over our `MATSEQCSRPERM` routines.)

Creating the parallel CSRPERM class, `MATMPICSRPERM`, is fairly trivial. Because the CSRPERM scheme only needs to be applied locally on each processor, the distributed CSRPERM class can be implemented by making only a trivial change to its parent `MATMPIAIJ` class. A `MATMPIAIJ` object is simply a collection of `MATSEQAIJ` objects that store the local portions of the matrix. Similarly, a `MATMPICSRPERM` is a collection of `MATSEQCSRPERM` objects: `MATMPICSRPERM` inherits from `MATSEQCSRPERM`, simply changing the types for the local matrix portions from `MATSEQAIJ` to `MATSEQCSRPERM`.

The implementations of the `MATCSRPERM` types are registered with PETSc inside the `MatRegisterAll` call that occurs at PETSc initialization. This makes it possible for existing PETSc codes to select our `MATCSRPERM` routines in place of the default `MATAIJ` ones at runtime by using the PETSc options database: specifying the command line option "-mat_type csrperm" will cause all matrices that determine their type from the options database to be of type `MATCSRPERM`.

## 5 Performance evaluation on the Cray X1

We have evaluated the performance of our `CSRPERM` implementation on Phoenix, a 512-MSP Cray X1 at the National Center for Computational Sciences at Oak Ridge National Laboratory. The X1 is organized into nodes consisting of four Multi-Streaming Processors (MSPs) that share a cache-coherent memory. Each MSP, in turn, consists of four tightly-coupled Single-Streaming Processors (SSPs) that share a 2MB L2 cache. Applications can be compiled to run in either SSP mode or MSP mode. In SSP mode, each SSP runs a separate MPI task. In MSP mode (the default), an MPI task occupies an entire MSP;

```
PetscErrorCode MatAssemblyEnd_SeqCSRPERM(Mat A, MatAssemblyType mode)
{
  PetscErrorCode ierr;
  Mat_SeqCSRPERM *csrperm = (Mat_SeqCSRPERM*) A->spptr;
  Mat_SeqAIJ     *a = (Mat_SeqAIJ*)A->data;
...
  a->inode.use = PETSC_FALSE;
  (*csrperm->AssemblyEnd_SeqAIJ)(A, mode);

  /* Now calculate the permutation and grouping information. */
  ierr = SeqCSRPERM_create_perm(A);
  PetscFunctionReturn(0);
}
```

Figure 11: The `assemblyend` method for the `MATSEQCSRPERM` matrix type.

the compiler handles automatic creation and synchronization of threads to use the resources of all four SSPs to perform the work in loops that can be "multistreamed".

In [4], we tested our `CSRPERM` implementation on a number of sparse matrices, some of which are summarized in Table 1. Table 2 shows the performance (in Mflops/s) of sparse matrix-vector multiply for the original CSR algorithm, versus the vectorizable CSRP and CSRPELL algorithms. The CSRP algorithm is roughly an order of magnitude faster than the original CSR algorithm. The CSRPELL algorithm can perform even better because the data rearrangement allows unit-stride memory access, though this comes at the expense of storing another copy of the matrix.

In addition to measuring the performance of our algorithms using a simple driver that loads a matrix and performs repeated mat-vecs, we have investigated its effect on the overall performance of some different applications. (In all cases we copy groups of rows to the ELL format.) Two of these applications are "toy" example codes that are provided with the PETSc distribution. The first is `ksp_ex2`, which solves a simple Laplace problem on a 2D grid using a five-point finite-difference stencil. The second is `snes_ex14`, which solves a solid fuel ignition problem on a 3D domain with a 7-point finite-difference stencil, using a Newton-Krylov method. In both of these examples, the default linear system solver is GMRES(30) preconditioned with ILU(0). Table 3 summarizes the wall-clock time required to run `ksp_ex2` on a $300 \times 300$ grid using one MSP, while Table 4 shows the wall-clock time required to run `snes_ex14` on a $32 \times 32 \times 32$ grid using one MSP. For `ksp_ex2`, the wall clock time required to reach a solution using the default solver is cut almost in half when using our vectorized matrix-vector multiplication kernel in place of the default one. This speedup comes from reducing the total time spent forming matrix-vector products from 219 seconds to 2 seconds. Almost all of the execution time when using the `CSRPERM` mat-vec is spent in applying the ILU(0) preconditioner, an operation that does not vectorize well on the X1. Because the `CSRPERM` kernel makes mat-vecs so inexpensive compared to the preconditioner application, we also tried using a point Jacobi (diagonal scaling) preconditioner. This preconditioner is much faster to apply, but is also much weaker than ILU(0), so many more iterations of GMRES(30) are required for convergence — thus work is shifted from applying the preconditioner to performing mat-vecs. When point Jacobi is used instead of ILU(0), the vectorized routine allows us to cut the total wall-clock time to only 37 seconds. (1423 seconds are required when point Jacobi is used without the `CSRPERM` mat-vec.) `snes_ex14` shows similar trends in performance, though the improvements are somewhat less dramatic.

Besides the two simple example codes, we also tested our `CSRPERM` routines in two full-scale application codes. The first code is PFLOTRAN, a parallel, fully-implicit code for multiphase groundwater (Darcy) flow and reactive transport coauthored by Richard Mills and Peter Lichtner (of Los Alamos National Laboratory). It uses a structured finite-volume mesh and employs a Newton-Krylov method to solve a nonlinear system of equations at each time step. We used the flow module, PFLOW, to solve a 3D flow and heat transport problem from the Nevada Test Site on a $95 \times 65 \times 50$ grid with three degrees of freedom per grid point. This is a very difficult problem involving convective plumes. We ran the problem in SSP mode using 512 SSPs. GMRES(30) is used for the inner

Table 1: Description of matrices

| Name | N | Nonzeros | Description |
|---|---|---|---|
| astro | 5706 | 60793 | Nuclear Astrophysics problem from Bradley Meyer |
| bcsstk18 | 11948 | 149090 | Stiffness matrix from Harwell Boeing Collection |
| 7pt | 110592 | 760320 | 7 point stencil in $48 \times 48 \times 48$ grid |
| 7ptb | 256000 | 7014400 | $4 \times 4$ blocks 7-pt stencil in $40 \times 40 \times 40$ grid |

Table 2: Performance (in Mflops/s) of sparse matrix multiply using CSR, CSRP and CSRPELL in PETSc.

| | SSP | | | MSP | | |
|---|---|---|---|---|---|---|
| Problem | CSR | CSRP | CSRPELL | CSR | CSRP | CSRPELL |
| astro | 26 | 163 | 311 | 14 | 214 | 655 |
| bcsstk18 | 28 | 315 | 340 | 15 | 535 | 785 |
| 7pt | 12 | 259 | 295 | 8 | 528 | 800 |
| 7ptb | 66 | 331 | 345 | 63 | 918 | 1085 |

| Method | Total | MatMult | PCApply |
|---|---|---|---|
| plain, block-Jacobi/ILU(0) | 451.3 | 218.9 | 227.6 |
| vec, block-Jacobi/ILU(0) | 235.8 | 1.6 | 229.5 |
| vec, point-Jacobi | 36.9 | 14.6 | 1.1 |
| plain, point-Jacobi | 1423.0 | 1400.0 | 1.1 |

Table 3: Performance of the ksp_ex2 code on one MSP. In the "Method" column, "plain" indicates that the default AIJ matrix-multiplication is used, while "vec" indicates that our CSRP algorithm is used. GMRES(30) is used as the iterative solver with the preconditioner indicated in the "Method" column. "Total" is the total time to solve the problem, while "MatMult" and "PCApply" indicate the amount of time spent performing matrix-vector multiplies and preconditioner applications, respectively. Timings are in seconds.

| Method | Total | MatMult | PCApply |
|---|---|---|---|
| plain, block-Jacobi/ILU(0) | 26.1 | 10.5 | 11.3 |
| vec, block-Jacobi/ILU(0) | 15.5 | 0.1 | 11.0 |
| vec, point-Jacobi | 5.3 | 0.7 | 0.1 |
| plain, point-Jacobi | 36.5 | 32.6 | 0.1 |

Table 4: Performance of the snes_ex2 code on one MSP. Column headings are as described for Table 3. Timings are in seconds.

solves in the Newton-Krylov iteration. Table 5 displays the performance we observed. When the linear systems are preconditioned with block Jacobi (one block per processor) using ILU(0) on the blocks, using the vectorized matrix-vector multiply routine cuts the total runtime by only a few minutes, or a little less than 20%. This is not surprising because the application spends relatively little of its time performing mat-vecs; getting appreciable speedup requires vectorizing the preconditioner as well. We tried shifting more of the work to matrix-vector products by using a point Jacobi preconditioner, but this ended up increasing the execution time because the preconditioner is poorly suited to this problem.

The second full-scale application code we tested our `CSRPERM` routines in is M3D [8], a three dimensional, unstructured finite-element magnetohydrodynamics code from Princeton Plasma Physics Laboratory (PPPL). M3D uses an operator-splitting approach and solves several elliptic equations at each time step. We used 16 MSPs to run a problem that follows the time evolution in a tokamak of an instability known as a resistive tearing mode—this involves magnetic field line reconnection and is driven by gradients in the toroidal plasma current. The linear systems are solved using GMRES with an additive Schwarz preconditioner with one level of overlap; ILU(3) is applied to each subdomain. Table 6 summarizes the performance observed. As in the PFLOTRAN case, our vectorized mat-vec reduces total execution time only slightly (by about 11%). The speedup for the matrix-vector multiplication is considerable (about 8.7 times), but a significantly larger fraction of the the work done by the application is in applying the preconditioner. This prompted us to try using a simple point Jacobi preconditioner with our vectorized mat-vec. Doing so does not reduce the execution time by any significant amount compared to using the non-vectorized mat-vec with the additive Schwarz/ILU(3) preconditioner. However, when point Jacobi is used, roughly half the execution time is spent performing orthogonalization with the GMRES basis. It might be the case that execution time could be improved considerably if we use a Krylov solver based on a short-term recurrence, such as TFQMR or BiCGSTAB, or if we simply use a more reasonable (i.e., smaller) basis size for restarted GMRES.

## 6 Summary and future directions

We have described our progress towards improving the performance of the PETSc toolkit for solving PDEs on the Cray X1. Our initial work has focused on improving the performance of sparse matrix-vector multiplication, an important kernel in iterative solvers. We have presented a simple, vectorizable algorithm that can calculate sparse matrix-vector products using the standard CSR storage format. The computation does involve non-unit stride access to the data, but nonetheless provides speedups that are typically an order of magnitude or better compared to the standard CSR multiply. At the cost of extra memory for a data copy, non-unit stride access can be eliminated, further improving performance. We have implemented a PETSc matrix type, `CSRPERM`, that is derived from the standard `AIJ` matrix type but uses our vectorizable matrix-multiplication algorithm. This type can be used with existing PETSc codes with little or no modification to those codes.

Performance tests indicate that our algorithm provided in `CSRPERM` speeds up sparse matrix-vector multiplication considerably, but experiments with full-fledged application codes indicate that further work is needed in other areas. Preconditioning of linear systems presents the biggest hurdle, as many popular preconditioners such as incomplete factorizations vectorize very poorly in their standard implementations. There are several possible means to speed up the triangular solves required to apply incomplete factorization preconditioners. Multicoloring can speed up both the triangular solves as well as the computation of the factorization. It has been used successfully with applications on the Earth Simulator [7], though it can slow convergence significantly for some problems. Another approach is to perform the forward and backward triangular solve in a block-recursive manner so that much of the work can be cast into the form of a matrix-vector multiply. Yet another possibility is to consider an expansion of the inverses of the factors and to take only a small number of terms in the series [10]; this can result in a very vectorizable solve operation with little degradation in accuracy. Of course, it may be the case that other preconditioners should be preferred over incomplete factorizations. Sparse approximate inverses [3, 5], for instance, may be preferable because once the preconditioner is constructed, applying it is as simple as performing a matrix-vector multiplication.

| Method | Total | MatMult | PCApply |
|---|---|---|---|
| plain, block-Jacobi/ILU(0) | 26.9 | 4.7 | 6.2 |
| vec, block-Jacobi/ILU(0) | 22.2 | 1.8 | 6.2 |
| vec, point-Jacobi | 33.7 | 10.3 | 0.3 |
| plain, point-Jacobi | 54.0 | 30.5 | 0.3 |

Table 5: Performance of the PFLOTRAN code on 512 SSPs. Column headings are as described for Table 3. Timings are in minutes.

| Method | Total | MatMult | PCApply |
|---|---|---|---|
| plain, additive Schwarz/ILU(3) | 42.0 | 7.8 | 17.1 |
| vec, additive Schwarz/ILU(3) | 37.3 | 0.9 | 17.1 |
| vec, point-Jacobi | 41.8 | 6.6 | 0.6 |
| plain, point-Jacobi | 94.3 | 57.3 | 0.6 |

Table 6: Performance of the M3D code on 16 MSPs. Column headings are as described for Table 3. Timings are in minutes.

## 7 Acknowledgments

We wish to thank Donald Spong of the Fusion Energy Division at ORNL for constructing the tearing mode test case for M3D and for his assistance in running the M3D performance tests.

## About the authors

Richard Tran Mills is a Computational Scientist at the Center for Computational Sciences (CCS) at Oak Ridge National Laboratory. He holds a Ph.D. in Computer Science with a specialization in Computational Science from the College of William and Mary. His research interests are in scalable numerical methods and their application to natural science problems. He can be reached at Oak Ridge National Laboratory, P.O. Box 2008 MS6008, Oak Ridge, TN 37831-6008, E-Mail: rmills@ornl.gov

Ed D'Azevedo is the acting group leader for the Computational Mathematics group in the Computer Science and Mathematics Division in the Oak Ridge National Laboratory (ORNL). He has a Ph.D. degree in Computer Science from the University of Waterloo, Canada and first came to ORNL under a postdoctoral research fellowship with the Oak Ridge Associated Universities in 1990. He has been involved in research in numerical linear algebra, optimal mesh generation and high performance computing with applications in fusion and astrophysics. He can be reached at Oak Ridge National Laboratory, P.O. Box 2008 MS6367, Oak Ridge, TN 37831-6367, E-Mail: dazevedoef@ornl.gov

Mark R. Fahey is a senior Scientific Application Analyst in the Center for Computational Sciences (CCS) at Oak Ridge National Laboratory. He is the current CUG X1-Users SIG chair. Mark has a Ph.D. in mathematics from the University of Kentucky. He can be reached at Oak Ridge National Laboratory, P.O. Box 2008 MS6008, Oak Ridge, TN 37831-6008, E-Mail: faheymr@ornl.gov

## References

[1] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.2.0, Argonne National Laboratory, August 2004. See also `http://www.mcs.anl.gov/petsc`.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

[3] Edmond Chow. Parallel implementation and practical use of sparse approximate inverses with *a priori* sparsity patterns. *Int. J. High Perf. Comput. Appl*, 15:56–74, 2001.

[4] Eduardo F. D'Azevedo, Richard T. Mills, and Mark R. Fahey. Vectorized sparse matrix multiply for compressed row storage format. *Lecture Notes in Computer Science*, 3514:99–106, 2005.

[5] M.J. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.*, 18(3):838–853, 1997.

[6] D. R. Kincaid and D. M. Young. The ITPACK project: Past, present and future. In G. Birkhoff and A. Schoernstadt, editors, *Elliptic Problem Solvers II Proc.*, pages 53–64, 1983.

[7] Kengo Nakajima and Hiroshi Okuda. Parallel iterative solvers for finite-element methods using a hybrid programming model on SMP cluster architectures. Technical Report GeoFEM 2003-003, RIST, Tokyo, 2003.

[8] W. Park, E. V. Belova, G. Y. Fu, and X. Z. Tang. Plasma simulation studies using multilevel physics models. *Physics of Plasmas*, 6:1796–1803, 1999.

[9] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, second edition, 2003.

[10] Henk van der Vorst. A vectorizable variant of some ICCG methods. *SIAM J. Sci. Statist. Comput.*, 3:350–356, 1982.