Progress Towards Optimizing the PETSc Numerical Toolkit on the Cray X-1

Richard Tran Mills\textsuperscript{1}, Ed D’Azevedo\textsuperscript{2}, Mark Fahey\textsuperscript{1}

\textsuperscript{1}National Center for Computational Sciences
\textsuperscript{2}Computer Science and Mathematics Division
Oak Ridge National Laboratory

Cray User Group Technical Meeting
May 19, 2005

This research supported by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL). This work was performed using resources at the National Center for Computational Sciences at ORNL, which is supported by the U.S. Department of Energy under Contract Number DE-AC05-00OR22725.
Deterministic solution of PDEs

- Many scientific codes simulate systems by solving PDEs.
- Typically:
  - Discretize system: Consider finite number of points
  - Obtain linear systems $Ax = b$
- Bulk of time spent solving large, sparse linear systems.
- Can solve with direct methods (Gaussian-elimination)
  - Guaranteed to find solution
  - But hard to scale to large systems, many processors
- Iterative methods are an increasingly popular alternative
  - Can scale to large problem sizes
  - Easy to parallelize
  - Require less time to find solution
Unfortunately…

- Modern iterative solver packages designed for scalar architectures!
  - Out-of-box performance is terrible!
- We describe ongoing work to provide vectorized PETSc kernels.

PETSc:
- Object-oriented framework for scalable solution of PDEs
- Several iterative (linear & nonlinear) solvers & preconditioners
- Seamless interface w/ other packages (e.g. SuperLU, Hypre)
- Shields user from complicated data structures, communication

- Initial work has focused on sparse matrix-vector multiply,
  a vital component of Krylov-subspace methods.
Outline

- Review sparse matrix storage formats, mat-vec algorithms
- Describe CSRPERM algorithm
  - With vectorization of CSR data in place
  - With rearrangement using ELLPACK storage
- Construction of CSRPERM matrix class into PETSc
  - Seamless integration to fully take advantage of PETSc
- Initial performance results on the X1
Compressed Sparse Row (CSR)

- CSR is most widely-used format for general sparse matrices
- Stores matrix in three arrays:
  - val: nonzero elements in row-by-row fashion
  - col_ind: column index of each element of val
  - row_ptr: points to beginning of each row in val

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

<table>
<thead>
<tr>
<th>val</th>
<th>11; 14; 21 22 24; 31 33 34 35; 43 44; 55</th>
</tr>
</thead>
<tbody>
<tr>
<td>col_ind</td>
<td>1 4; 1 2 4; 1 3 4 5; 3 4; 5</td>
</tr>
<tr>
<td>row_ptr</td>
<td>1 3 6 10 12 13</td>
</tr>
</tbody>
</table>

- Mat-vec proceeds directly through val, operating row-by-row.
- Poor performance on vector machines b/c of short row vectors
  - 1st order star-type FD stencil: 5 elements per row in 2D, 7 elements in 3D
If all rows have similar # nonzeros, can use ELLPACK format

Uses two N x NZMAX arrays constructed by:

- Shifting all nonzeros left
- Columns of shifted “matrix” stored consecutively in val
- Corresponding col_ind array stores column indices

Mat-vecs proceed along columns of val
Long vectors + regular access yields good compiler vectorization
Jagged Diagonal (JAD) storage eliminates zero padding of ELL.

To construct:

- Permute matrix, ordering rows by decreasing number of nonzeros
- First JAD: leftmost nonzeros of row 1, row2, etc. of $PA$
- Second JAD: next nonzeros from row 1, row2, etc.

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

$$\begin{array}{c}
\text{jdiag} \\
\text{col_ind} \\
\text{jd_ptr} \\
\text{perm}
\end{array} = \begin{pmatrix}
31 & 21 & 11 & 43 & 5; \\
33 & 22 & 14 & 44; \\
34 & 24; \\
35 \\
1 & 1 & 1 & 3 & 5; \\
3 & 2 & 4 & 4; \\
4 & 4; \\
5 \\
1 & 6 & 10 & 12 \\
3 & 2 & 1 & 4 & 5
\end{pmatrix}$$

- Mat-vecs proceed along jagged diagonals; yields long vector lengths
- Significant memory traffic to repeatedly read/write result vector $y$
Like JAD, sort (permute) rows based on # nonzeros.

Construct groups of rows w/ same # nonzeros.

Mat-vec computed one group at a time:
- Performs mat-vec for a group in same manner as ELLPACK
- No zero padding b/c of sorting

Reduced memory bandwidth requirements compared to JAD.

Can leave CSR data in place (CSRP):
- Only need O(N) extra storage for permutation
- Irregular memory access to val array

Or, can copy groups into ELLPACK format (CSRPELL):
- Better memory access pattern
- But storage requirements doubled
Conceptual comparison between JAD and CSR P

ROWS

ROWS

NONZEROS

NONZEROS

JAGGED DIAGONAL

CSR WITH PERMUTATION
Creating a CSRPERM matrix class for PETSc

- PETSc is written in C, but uses an object-oriented design:
  - Has its own function tables, dispatch mechanism
  - Employs data encapsulation, polymorphism, inheritance
- All PETSc objects are derived from an abstract base type
  - Mat is the base matrix type
  - MATAIJ is the standard CSR-format instantiation

- We seamlessly integrate support for our CSRP algorithm into PETSc, creating a CSRPERM matrix type derived from AIJ.
- We inherit most methods from AIJ; only a few select methods must be overridden.
Matrix creation method

- In PETSc, a Mat object `A` is built into a particular type by
  `MatSetType(Mat mat, MatType Type)`
- If `Type` is MATSEQCSRPERM, then PETSc calls our internal routine:

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

- Line 4 builds an empty MATSEQAIJ matrix.
- Line 5 converts that to object to our MATSEQCSRPERM type.
MatConvert Routine

```c
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);
}
```

- Lines 7-8 allocate CSRPERM data structure, stash it in `spptr`.
- Lines 12-16 set pointers for AIJ methods we override.
Assembly of the CSRP matrix

- In PETSc, `assemblyend` finalizes construction of matrix data structure.
- Creating `CSRPERM` proceeds from `AIJ` data structure, so use `AIJ` `assemblyend` and then proceed from there.

```c
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(ierr);
}
```
What I’ve shown so far is for the sequential CSRPERM instantiation.

Implementing the parallel MATMPI CSRPERM class is trivial!

MPIAIJ is simply a collection of SeqAIJs storing local matrix portions

Similarly, MPICSRPERM a collection of SeqCSRPERMs:

- MPICSRPERM inherits from MPIAIJ;
  changes the type for local mats from SeqAIJ to SeqCSRPERM.
So why bother writing all this glue code?

- Use CSRP kernels without modification to existing codes
  - Register CSRPERM class with PETSc
  - Use PETSc’s options database to select appropriate routines: “-mat_type csrperm”
  - Use options database to set CSRPERM options (e.g., copy groups to ELLPACK format or not)

- Get CSRPERM accepted into the official PETSc source
  - Now a supported matrix class
  - Available in petsc-dev now; should be in next public release
### Performance: Sparse mat-vec

<table>
<thead>
<tr>
<th>Name</th>
<th>N</th>
<th>Nonzeros</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astro</td>
<td>5706</td>
<td>60793</td>
<td>Nuclear astrophysics problem</td>
</tr>
<tr>
<td>bcsstk18</td>
<td>11948</td>
<td>80519</td>
<td>Stiffness matrix from Harwell-Boeing library</td>
</tr>
<tr>
<td>7pt</td>
<td>110592</td>
<td>760320</td>
<td>7-pt stencil in 48 x 48 x 48 grid</td>
</tr>
<tr>
<td>7pt_blk</td>
<td>256000</td>
<td>7014400</td>
<td>4x4 blocks 7-pt stencil in 40 x 40 x 40 grid</td>
</tr>
</tbody>
</table>

**Diagram:**
- **astro**: Graph showing a sparse matrix with 60793 nonzeros.
- **bcsstk18**: Graph showing a sparse matrix with 80519 nonzeros.
Performance: Sparse mat-vec

<table>
<thead>
<tr>
<th>Name</th>
<th>N</th>
<th>Nonzeros</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astro</td>
<td>5706</td>
<td>60793</td>
<td>Nuclear astrophysics problem</td>
</tr>
<tr>
<td>bcsstk18</td>
<td>11948</td>
<td>80519</td>
<td>Stiffness matrix from Harwell-Boeing library</td>
</tr>
<tr>
<td>7pt</td>
<td>110592</td>
<td>760320</td>
<td>7-pt stencil in 48 x 48 x 48 grid</td>
</tr>
<tr>
<td>7pt_blk</td>
<td>256000</td>
<td>7014400</td>
<td>4x4 blocks 7-pt stencil in 40 x 40 x 40 grid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
<th>SSP</th>
<th>MSP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CSR</td>
<td>CSRP</td>
</tr>
<tr>
<td>astro</td>
<td>26</td>
<td>163</td>
</tr>
<tr>
<td>bcsstk18</td>
<td>28</td>
<td>315</td>
</tr>
<tr>
<td>7pt</td>
<td>12</td>
<td>259</td>
</tr>
<tr>
<td>7pt_blk</td>
<td>66</td>
<td>331</td>
</tr>
</tbody>
</table>

Performance of sparse mat-vec in MFlops/s
Performance: PETSc example codes

Run two PETSc examples on 1 MSP:

- **ksp_ex2**: Solves 2D Laplace problem w/ 5-pt FD stencil, 300x300 grid

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>MatMult</th>
<th>PCApply</th>
</tr>
</thead>
<tbody>
<tr>
<td>plain, GMRES+ILU(0)</td>
<td>451.3</td>
<td>218.9</td>
<td>227.6</td>
</tr>
<tr>
<td>vec, GMRES+ILU(0)</td>
<td>235.8</td>
<td>1.6</td>
<td>229.5</td>
</tr>
<tr>
<td>vec, GMRES+Jacobi</td>
<td>36.9</td>
<td>14.6</td>
<td>1.1</td>
</tr>
<tr>
<td>plain, GMRES+Jacobi</td>
<td>1423</td>
<td>1400.0</td>
<td>1.1</td>
</tr>
</tbody>
</table>

- **snes_ex14**: 3D fuel ignition via Newton-Krylov, 7pt FD, 32x32x32 grid

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>MatMult</th>
<th>PCApply</th>
</tr>
</thead>
<tbody>
<tr>
<td>plain, GMRES+ILU(0)</td>
<td>26.1</td>
<td>10.5</td>
<td>11.3</td>
</tr>
<tr>
<td>vec, GMRES+ILU(0)</td>
<td>15.5</td>
<td>0.1</td>
<td>11.0</td>
</tr>
<tr>
<td>vec, GMRES+Jacobi</td>
<td>5.3</td>
<td>0.7</td>
<td>0.1</td>
</tr>
<tr>
<td>plain, GMRES+Jacobi</td>
<td>36.5</td>
<td>32.6</td>
<td>0.1</td>
</tr>
</tbody>
</table>
Performance: PFLOTRAN

- PFLOTRAN: Parallel, fully implicit, multiphase groundwater flow and transport code; coauthored w/ Peter Lichtner at LANL
- Run 3D flow + heat transport problem from NTS on 512 SSP’s
- 95 x 65 x 50 grid, 3 degrees of freedom per node

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>MatMult</th>
<th>PCApply</th>
</tr>
</thead>
<tbody>
<tr>
<td>plain, GMRES+ILU(0)</td>
<td>26.9</td>
<td>4.7</td>
<td>6.2</td>
</tr>
<tr>
<td>vec, GMRES+ILU(0)</td>
<td>22.2</td>
<td>1.8</td>
<td>6.2</td>
</tr>
<tr>
<td>vec*, GMRES+Jacobi</td>
<td>33.7</td>
<td>10.3</td>
<td>0.3</td>
</tr>
<tr>
<td>plain, GMRES+Jacobi</td>
<td>54.0</td>
<td>30.5</td>
<td>0.3</td>
</tr>
</tbody>
</table>
Performance: M3D

- M3D: 3D resistive MHD code from PPPL.
- Run on 16 MSPs w/ on a tearing-mode case.

<table>
<thead>
<tr>
<th>Method</th>
<th>total</th>
<th>MatMult</th>
<th>PCApply</th>
</tr>
</thead>
<tbody>
<tr>
<td>plain, GMRES+ILU(3) on subdomains</td>
<td>42.0</td>
<td>7.8</td>
<td>17.1</td>
</tr>
<tr>
<td>vec, GMRES+ILU(3) on subdomains</td>
<td>37.3</td>
<td>0.9</td>
<td>17.1</td>
</tr>
<tr>
<td>vec, GMRES+Jacobi</td>
<td>41.8</td>
<td>6.6</td>
<td>0.6</td>
</tr>
<tr>
<td>plain, GMRES+Jacobi</td>
<td>94.3</td>
<td>57.3</td>
<td>0.6</td>
</tr>
</tbody>
</table>

- Can’t improve time w/ Jacobi, but note that 21-22 minutes spent in GMRES orthogonalization!
- PPPL currently uses GMRES basis size of 1000!
- Might be a win if we use TFQMR, Bi-CGSTAB… or simply a more reasonable GMRES basis size!
Summary and Future Directions

- Presented the CSRP mat-vec algorithm
  - Promotes long vector lengths
  - Can work well with CSR data left in-place
  - Implemented CSRPERM matrix type in PETSc

Preconditioning still presents a big hurdle:

- Could try to speed up triangular solves for ILU
  - Multicoloring can work, but degrades preconditioner quality
  - Block-recursive formulation yielding series of mat-vecs
  - Take first few terms of Neumann expansion of factorization

- Don’t use incomplete factorizations?
  - Sparse approximate inverses
  - Polynomial preconditioners