A Fast Poisson Solver for PETSc

B. A. Grierson* and Hong Zhang
Department of Applied Physics and Applied Mathematics
Columbia University, New York, NY 10027
(Dated: May 9, 2007)

I. INTRODUCTION

This small monograph describes, in detail, the implementation of a Fast Poisson Solver, using different methods for solving the resulting tridiagonal system.[1]

II. THE EQUATION TO SOLVE

We wish to solve Poisson's Equation ($\epsilon_0 \equiv 1$) on a 2D domain. We will consider the domain $x \times y = [0, 2\pi) \times [0, 1]$. The equation is

$$\begin{aligned} \nabla^2 \Phi &= -\rho \\ \frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} &= -\rho(x, y) \\ \Phi(x, 0) = \Phi(x, 1) &\equiv 0 \quad (B.C.) \end{aligned} \tag{1}$$

We wish to solve equation(1) for Φ, given ρ, as fast as possible.

III. THE FFT METHOD

Because we are working on a periodic domain in x, we can use the FFT to convert our 2D equation into a system of 1D equations. This is done as follows.
Define our domain and grid quantities.

- M_x, M_y: Number of grid points in x and y, respectively.
- $\Delta x = 2\pi/M_x$: The x grid spacing, which doesn't include the 2π boundary point.
  The x axis is $m\Delta x$, where $m \in [0, M_x - 1]$.
- $\Delta y = 1/(M_y - 1)$: The y grid spacing.
  The y axis is $l\Delta y$, where $l \in [0, M_y - 1]$.

The discrete Fourier transform (and inverse transform) of a function g is given by

$$\begin{aligned} g(k, y) &= \frac{1}{M_x} \sum_{m=0}^{M_x - 1} g(m, y) e^{-imk\Delta x} \\ g(m, y) &= \sum_{k=0}^{M_x - 1} g(k, y) e^{imk\Delta x} \end{aligned} \tag{2}$$

If g is a real function, then $g(M_x - m) = [g(m)]^*$.
Differential operators are given by

$$\begin{aligned} \left.\frac{\partial g}{\partial x}\right|_k &\approx \frac{1}{2\Delta x}\big[g(k+1) - g(k-1)\big] \to -\frac{i}{\Delta x} g(m) \sin(m\Delta x) \\ \left.\frac{\partial^2 g}{\partial x^2}\right|_k &\approx \frac{1}{(\Delta x)^2}\big[g(k+1) - 2g(k) + g(k-1)\big] \to \frac{2}{(\Delta x)^2} g(m)\big[\cos(m\Delta x) - 1\big] \end{aligned} \tag{3}$$

We now have the tools to convert (1) into a form which can be solved for quickly. Our equation now becomes

$$\begin{aligned}
\frac{2}{(\Delta x)^2}\Phi(m,l)\big[\cos(m\Delta x)-1\big]+\frac{\Phi(m,l+1)-2\Phi(m,l)+\Phi(m,l-1)}{(\Delta y)^2} &= -\rho(m.l) \qquad (4)\\
\Phi(m,l+1)+\left\{\frac{2(\Delta y)^2}{(\Delta x)^2}\big[\cos(m\Delta x)-1\big]-2\right\}\Phi(m,l)+\Phi(m,l-1) &= -(\Delta y)^2\rho(m.l)\\
\Phi(m,0)=\Phi(m,M_y) &\equiv 0 \quad (B.C.)
\end{aligned}$$

For each m, this is a 1D tridiagonal system in the y direction. The first and last equations ($l=0, l=M_y-1$) are not solved. They are set to zero.

Once the solution to the tridiagonal system is obtained (in Fourier space), we back-transform to recover $\Phi(x,y)$.

IV. THE IMPLEMENTATION OF THE METHOD

A. Setup

We will solve (1) in parallel with PETSc to manage the distributed arrays and communication. We will also use FFTW to perform the FFTs.

PETSc includes objects for managing Distributed Arrays (DAs) for parallel computing. This allows us to set the array distribution in the desired manner. We break the y domain across processors, such that our square grid consists of a series of p horizontal bands, width M_x (Fig.1).

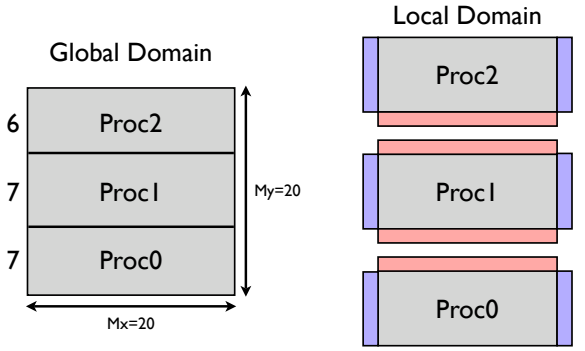

FIG. 1: The global and local domains for our problem, shown for a $(M_x \times M_y) = (20 \times 20)$ grid. 3 processors.

This distribution allows us to take the FFT in the 'x' periodic direction, unbroken by processor boundaries. But, this distribution requires us to form our tridiagonal system *across* processors.

After taking the Fourier Transform of ρ, we define our tridiagonal system, for a **fixed** m, as follows:

$$\mathcal{A}=\begin{bmatrix} a_0 & c_0 & 0 & 0 & \dots & 0\\ b_1 & a_1 & c_1 & 0 & \dots & 0\\ 0 & b_2 & a_2 & c_2 & \dots & 0\\ . & . & . & . & \dots & .\\ 0 & \dots & & & \dots & c_{M_y-2}\\ 0 & \dots & & b_{M_y-1} & a_{M_y-1} & \end{bmatrix}_{M_y\times M_y} \qquad \rho=\begin{bmatrix}\rho_0\\ \rho_1\\ \rho_2\\ .\\ \rho_{M_y-2}\\ \rho_{M_y-1}\end{bmatrix}_{M_y\times 1} \qquad (5)$$

where the matrix $\mathcal{A}$ and the RHS are stored as individual vectors distributed in the y direction. The vectors are **A**, **B**, **C**, ρ where **A**=$\{a_l\}_{M_y}$, **B**=$\{b_l\}_{M_y}$, **C**=$\{c_l\}_{M_y}$, $\rho=\{\rho_l\}_{M_y}$.

Therefore, for a **fixed** m,

$$\begin{aligned}
a_l &= \frac{2(\Delta y)^2}{(\Delta x)^2)}\big[\cos(m\Delta x)-1\big]-2+(0.0)i \qquad (6)\\
b_l &= 1.0+(0.0)i\\
c_l &= 1.0+(0.0)i
\end{aligned}$$

and ρ is replaced by the (complex) FFT of itself. This system is then solved.

To summarize, the method is given in PETSc C pseudo-code:

```
// Fast Poisson Solver
PetscErrorCode = FPS( Vec Rho, Vec Pot)
{
  Vec    A,B,C,D,S;

  // Set the off-diagonal 1D vectors.
  B = 1.0 + 0.0*PETSC_i;
  C = 1.0 + 0.0*PETSC_i;

  // Replace 2D rho with its FFT
  GetArray(Rho,&rho);
  for(j=0; j<My; j++){
    rho[j][*] = FFT_FORWARD(rho[j][*]);
  }
  RestoreArray(Rho,&rho);

  // Set up and solve the  tridiagonal system.
  GetArray(Rho,&rho);
  GetArray(Pot,&pot);
  tmpA = 2.0*pow(dy/dx,2)

  for(m=0; m<Mx/2; m++){

    // Set the diagonal vector for the given m.
    GetArray(A,&a);
    for(j=0; j<My; j++) a[j] = tmpA*(cos(m*dx)-1.0) - 2.0;
    RestoreArray(A,&a);

    // Set the RHS for the given m.
    GetArray(D,&d);
    for(j=0; j<My; j++) d[j] = dy*dy*(RealPart(rho[j][m]) + ImaginaryPart(rho[j][m])*PETSC_i);
    RestoreArray(D,&d);

    // Solve the tridiagonal system, returning S, the solution.
    SolveSystem(A,B,C,D,S);

    GetArray(S,&s);
    for(j=0; j<My; j++){
      Pot[j][m] = s[j]
    }
    RestoreArray(S,&s);
  }  // Done with the solve.
  RestoreArray(Rho,&rho);
  RestoreArray(Pot,&pot);


  // Replace Pot with its inverse FFT
  GetArray(Pot,&pot);
  for(j=0; j<My; j++){
    Pot[j][*] = FFT_BACKWARD(Pot[j][*]);
  }
  // Done
}
```

B. The Solver

In the above pseudo-code, the call to `SolveSystem(...)` is overwhelmingly the most expensive part. The FFTs take essentially no time. One method to solve the system is to scatter the **entire system** to one processor, then use a sequential LU factorization and back-substitution. For small system sizes, or small number of processors, this is the fastest method.[2]

For large system sizes, another method[1] has been shown to be fast and efficient. This method (called the Parallel Partition Method) is also an exact solver.

We break the matrix operator into a block-tridiagonal matrix plus the residual off-diagonal elements on processor boundaries, illustrated in (Fig.2). The system is decomposed as follows, where calligraphy characters are matrices, bold are vectors, and $\mathcal{I}$ is the identity matrix.

$$\begin{aligned} \mathcal{A}\mathbf{x} &= \mathbf{b} \qquad (7)\\ \mathcal{A} &= \tilde{\mathcal{A}} + \Delta\mathcal{A} \\ &= \tilde{\mathcal{A}} + \mathcal{V}\mathcal{E}^T \end{aligned}$$

$$\begin{aligned} \mathbf{x} &= \mathcal{A}^{-1}\mathbf{b} \qquad (8)\\ &= (\tilde{\mathcal{A}} + \mathcal{V}\mathcal{E}^T)^{-1}\mathbf{b} \\ &= \tilde{\mathcal{A}}^{-1}\mathbf{b} - \tilde{\mathcal{A}}^{-1}\mathcal{V}(\mathcal{I} + \mathcal{E}^T\tilde{\mathcal{A}}^{-1}\mathcal{V})^{-1}\mathcal{E}^T\tilde{\mathcal{A}}^{-1}\mathbf{b} \end{aligned}$$

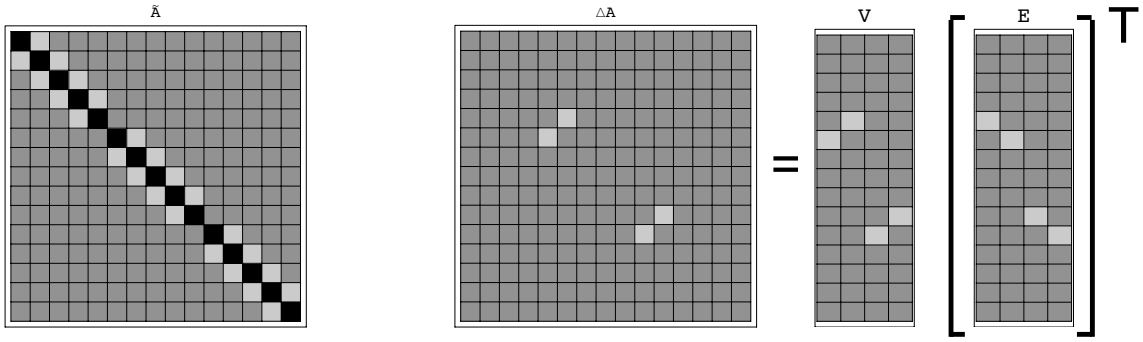

FIG. 2: The partitioning of the original operator $\mathcal{A}$ into $\mathcal{A} = \tilde{\mathcal{A}} + \Delta\mathcal{A} = \tilde{\mathcal{A}} + \mathcal{V}\mathcal{E}^T$. The given example is for a 15×15 system on $p = 3$ processors.

The solution procedure is summarized as follows for the general system $\mathcal{A}\mathbf{x} = \mathbf{b}$, where the left-aligned equations are solves, and the right-aligned equations are algebraic.

$$\begin{aligned} \tilde{\mathcal{A}}\tilde{\mathbf{x}} &= \mathbf{b} && (9)\\ \tilde{\mathcal{A}}\mathcal{Y} &= \mathcal{V} && (10)\\ \mathbf{h} &= \mathcal{E}^T\tilde{\mathbf{x}} && (11)\\ \mathcal{Z} &= \mathcal{I} + \mathcal{E}^T\mathcal{Y} && (12)\\ \mathcal{Z}\mathbf{y} &= \mathbf{h} && (13)\\ \Delta\mathbf{x} &= \mathcal{Y}\mathbf{h} && (14)\\ \mathbf{x} &= \tilde{\mathbf{x}} - \Delta\mathbf{x} && (15) \end{aligned}$$

The equation for $\mathbf{y}$ (13), called the *interface system* is a solve for a non-tridiagoanl system. The other solves (9,10) are tridiagonal, hence we can use the standard LU method to solve these equations.

The equation for $\mathbf{y}$ is solved using a dense decomposition and back-substitution. This is still fast for a small number of processors, because the interface system size is $2p - 2$.

In practice, this method requires storage of the three vectors (length M_y) which make up $\mathcal{A}$, the $2p - 2$ vectors (length M_y) which make up $\mathcal{V}$, the full matrix $\mathcal{Z}$ (size $2p - 2 \times 2p - 2$), and the vector $\mathbf{h}$ (length $2p - 2$).

The communication required depends on where we want the solution to take place. One can either scatter the values of $\mathcal{Z}$ and $\mathbf{h}$ to the rank=0 process, solve, and scatter the solution vector $\mathbf{y}$ back to all processes, or scatter the values of $\mathcal{Z}$ and $\mathbf{h}$ to all processes, and solve the system redundantly.

Sending the system to the rank=0 process requires $4p^2 - 10p + 6$ Send/Recv calls for $\mathcal{Z}$ and $2p - 3$ Send/Recv calls for $\mathbf{h}$, where the Send calls are from all processes except rank=0, and the Recv calls are all made by rank=0. We then scatter back only the solution ($2p - 3$ Send/Recv calls).

Sending the full matrix $\mathcal{Z}$ and vector $\mathbf{b}$ to all processes requires overlapping communication. The rank=0 and rank=$p - 1$ processes each send $2p - 2$ values, and receive $4p^2 - 10p + 6$ values for $\mathcal{Z}$ (send 1 and receive $2p - 3$ values for $\mathbf{h}$). The interior processors each send $2(2p - 2)$ values and recieve $4p^2 - 12p + 8$ values for $\mathcal{Z}$ (send 2 and receive $2p - 4$ values for $\mathbf{h}$). The system is then solved redundantly on each processes, and no more communication is required.

We will test the performance of these two methods, to determine which is faster.

V. SUMMARY OF THE PETSC IMPLEMENTATION

A. Managing The Data (Vecs)

The current (Sat April 28, 2007) implementation of the Fast Poisson solver is done in PETSc on a 2D distributed array (DA) object. The variables Φ and ρ are initialized into `Vecs`. A 1D DA is created to manage the storage and manipulation of the vectors which make up the full tridiagonal system, which is distributed across processors.

The Fast Poisson Solve routine should have the following characteristics:

- Input:
  The known `Vec` varaible, ρ.
  An empty `Vec` variable, Φ to hold the solution.
  A user context containing the information about the grids and domain (size, rank, DA, $\Delta x, \Delta y, \ldots$).
- Output: The filled `Vec`

B. The Solver

The solver will need to know some specific information at runtime. First and foremost, the method of solve. Do we want to use the Parallel Partition Method or the ScatterToZero method? The user will either need to be informed about when to use which (system size, number of processors, etc. . .), or have `PETSC_DECIDE` as an option.

The memory needed for the solve. If the solve will need to be done many times ($\sim 10^6$), then certain variables will need to be placed in a some type of solver object for efficient allocation of memory. We don't want to call `PetscMalloc(),PetscFree()` a million times.

For the ScatterToZero method, we need the scatter context, and the sequential vectors created with `VecScatterCreateToZero()`. We then solve the system as many times as needed. Then `VecDestroy()` when we're done solving the system a million times. We don't want to call these routines for every solve.

For the Parallel Partition Method, we need to allocate memory using `PetscMalloc()` for both the matrix $\mathcal{Z}$ and the vector $\mathbf{h}$. We also need to create the array of 1D DA `Vecs` for the matrix $\mathcal{V}$

C. Example Implementation

I have implemented this entire solver into my simulaiton. Here is the summary of how I do it.

```
Compile and link the routines.
Pass the type of solver I want to use as a runtime option in the makefile (-SeqTri or -ParTri).
```

1. `Allocate the memory and needed objects for the type of solver chosen at runtime.`
2. `Place all required variables into the user context.`
   `Vec AZero, BZero, ... from VecScatterCreateToZero for -SeqTri`
   `PetscScalar **Z, *h, ... Vec *V, .... for -ParTri.`

- `BEGIN SIMULATION LOOP` (for a few million iterations).
- `Form the RHS 2D Vec, and pass it into the FastPoissonSolver (FPS)`
- `Do forward FFTs`
- `Use the solver` $M_x/2$ times.
  - If this is a uniprocessor run, use the sequential LU decomp and back sub.
  - If this is a `-SeqTri` run, `SCATTER_FORWARD` for all the `Vecs` on the 1D DA, use the sequential LU decomp and back sub on the rank=0 process, then `SCATTER_REVERSE` just the solution.
  - If this is a `-ParTri` run, zero the entries to `Vec *V`, then use the parallel solver with the BlockTriLUDecomp and BlockTriLUBackSub, as well as the DenseLUDecmp, and DenseLUBackSub routines, adapted from *Numerical Recipes in C*.
- `Do backwards FFTs`
- `Make a time step.`
- `END SIMULATION LOOP`

3. `Free the memory and objects created  based on runtime option.`

The above skeleton is the general simulation in which a fast, memory efficient, Fast Poisson Solver is required for my simulation, as well as (I'm sure) many other people requesting fast Poisson/Tridiagonal solvers.

VI. SUMMARY

In conclusion, it is worthwhile to develop a general package for solving Poisson's equation, and a general tridiagonal system in parallel, without making any assumptions about symmetry or diagonal dominance of the system $\mathcal{A}\mathbf{x} = \mathbf{b}$. The creation of the two routines `DAFPS` for the Fast Poisson Solve, and `DATriSolver` for the tridiagonal solver will be worthwhile for the PETSc libraries.

VII. PERFORMANCE

Presented in the following figures are the scaling results for the Distributed Array Fast Poisson Solver (`DAFPS`). These results are for the full solve, not just the tridiagonal solve. We input a set potential, and differentiate to acquire the charge density. The potential and charge are set and calculated as

$$\begin{aligned} \Phi(x,y) &= \sin(x)\sin(2\pi y) \qquad x \in [0,2\pi),\ y \in [0,1] \\ \rho(j,i) &= -\frac{\Phi(j,i+1) - 2\Phi(j,i) + \Phi(j,i-1)}{\Delta x} - \frac{\Phi(j+1,i) - 2\Phi(j,i) + \Phi(j-1,i)}{\Delta y} \end{aligned} \tag{16}$$

We then input ρ into the solver, and calculate Φ_{sol}. Then the $2-norm$ error of $\Phi - \Phi_{sol}$ is computed. Because this is an exact solve, the error is $\mathcal{O}(10^{-13})$ for a $(M_x \times M_y) = (100 \times 100)$ grid.

Before presenting the results, it should be stressed that it will be difficult to beat the sequential, uniprocessor solution for the tridiagonal system. The grid size must be much greater than the number of processors ($M_y >> p$). This is because of different reasons for different methods.

1. The Sequential Solver
   - The scatter to zero requires $(1-1/p)M_y(\times 4$ `Vecs`$)$ calls to Send from all but the rank=0 process (and an equal number of receives for the rank=0 process), as well as the back-scatter of $(1-1/p)M_y$ values of the solution.
   - the **full** solution must then performed on the rank=0 process.
   - The total cost is the cost of the uniprocess solve plus the cost of the communcation.
2. The Parallel Partition Solver.
   - The solution of (9,10) are done in parallel, but they first require the parallel LU inverse of $\tilde{\mathcal{A}}$ (which is only performed once), **plus** $(2p-2)+1$ back-substitutions for **b** and **each column** of $\mathcal{V}$.

- Then communication is performed for $\mathbf{h}$ and $\mathcal{Z}$.
- Then the dense LU decomposition and back-substitution.

We can get good speedup for the Fast Poisson Solve, because the FFTs are done in parallel. Therefore, there is a tradeoff between faster FFTs (more processors), and faster tridiagonal solution (fewer processors) as will be seen in the results. Either way, a larger enough system will always solve faster for more processors. The question is: When do we use which solver?

For the `DAFPS` solver as a part of a larger simulation, that will be up to the user. Say a simulation is to be run on 10 processors. We aim to determine which solver to use for the given grid size. If the `DAFPS` solver is a small-cost portion of the simulation, than the advantage of more processors for some routines might outweigh the disadvantage of slowing down `DAFPS`.

A. Scaling

There are two types of scaling (Fig.3) which are standard to use in parallel computing. These are *Strong* and *Weak* scaling.

Strong scaling is the change of 'wall-clock' execution time for a fixed problem size, and increasing number of processors. Good strong scaling has a $t \sim 1/p$ curve. In other words, throw more processors at a given job, and get it done faster. The measure of performance for strong scaling is the *speedup*. *Speedup* is the ratio of the execution time to the uni-processor execution time $S_p = t_1/t_p$. Perfect speedup on 3 procesors is $S_3 = 3.0$, i.e. the job executed in 1/3 the time it would have on a uni-processor machine.

Weak scaling represents the change (or lack of) in wall clock time as both the number of processors and problem size are increased in proportion, keeping the *same problem size* on each processor. Good weak scaling has a $t \sim cnst$ curve. If we have a 1D problem, and $N = 100$ points in 'x'. For three processors, we change N to 300 points, and re-run the problem. The execution time should not change if the code exhibits weak scaling.

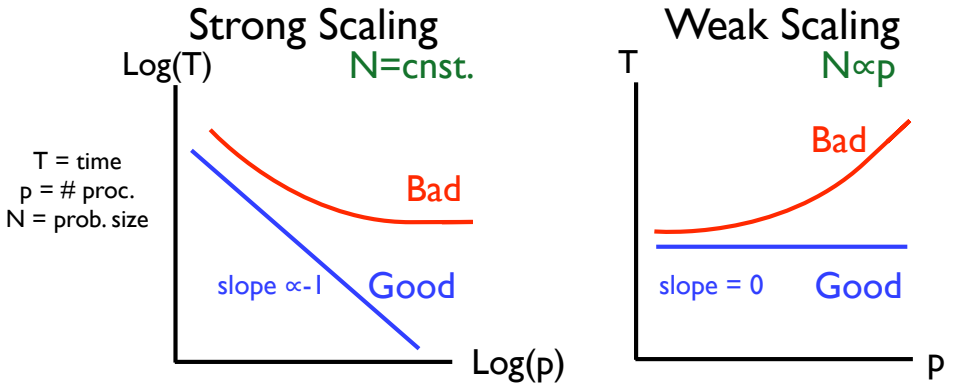

FIG. 3: Two types of scaling.

B. Computer

The `DAFPS` code has been run on a multi-processor machine Orion DT-12, with12 Transmeta Efficeon TM8000 1.2GHz processors. The available memory is 12 DIMMS (one per node), 1GB each (DDR333/PC2700).

All the results presented here use the PETSc pre-loading calls `PreLoadBegin(), PreLoadEnd()`.

C. Strong Scaling and Speedup

Presented below (Fig.4) are the *Strong Scaling* results for both the 'ScatterToZero' method, and the Parallel Partition Method (v2.0, v2.1) on a $M_x \times M_y$ grid ($M_x = M_y$). The three runs below display the transition from poor to 'acceptable' performance for three different grid sizes ($M_y = 1000, 2000, 3000$). We can see from the job-time

bar charts that the first version of the parallel solver (v2.0) was very slow in forming the $\mathcal{Z}$ matrix. This has been optimized with non-blocking MPI calls, and using PETSc to factor and solve the interface system $\mathcal{Z}\mathbf{y} = \mathbf{h}$.

The problem now is the formation of the $\mathcal{V}$ matrix, and the solution of the block-tridiagonal systems $\tilde{\mathcal{A}}\tilde{\mathbf{x}} = \mathbf{d}$ and $\tilde{\mathcal{A}}\mathcal{Y} = \mathcal{V}$. First, $\tilde{\mathcal{A}}$ is LU-factored in a block-diagonal manner. Then the factored $\tilde{\mathcal{A}}$ is us to back-sub solve for $\mathbf{d}$ and each column of $\mathcal{V}$ (labeled as `DABTLUBSD` and `DABTLUBSV` in Fig.7).

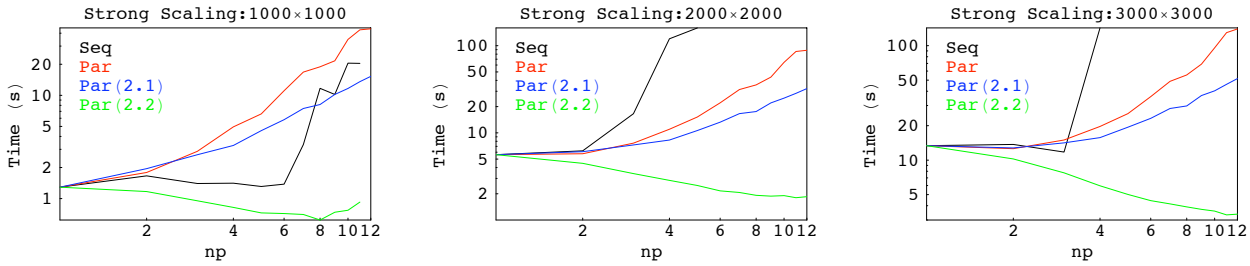

FIG. 4: Strong Scaling Results for the ScatterToZero Sequential, Parallel Partition (solve interface system on all processors), and optimized (v2.1, v2.2) version of the Parallel Partition method.

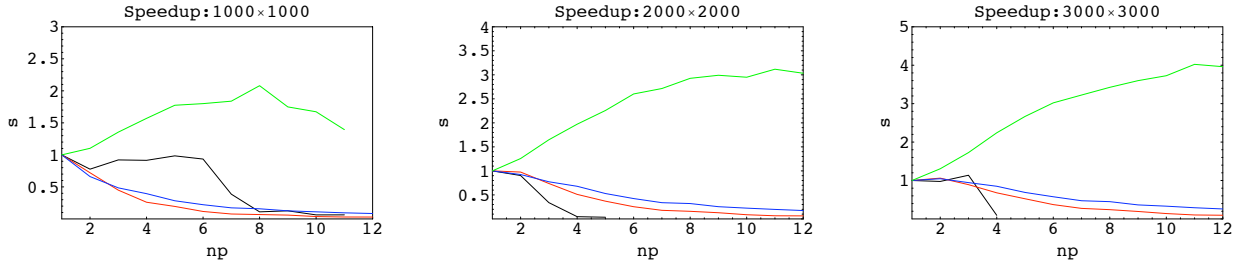

FIG. 5: Speedup for the ScatterToZero Sequential, Parallel Partition (solve interface system on all processors), and optimized (v2.1, v2.2) version of the Parallel Partition method.

D. Weak Scaling

Presented below are the Weak scaling results for the entire Fast Poisson Solve. We set a base ($p = 1$) grid size, and scale the grid with the number of processors. Increasing the grid size as $1000p$ scales with the solver (which is 1D across processes), whereas increasing the grid size as $1000\sqrt{p}$ scales with the total problem. Both are included.

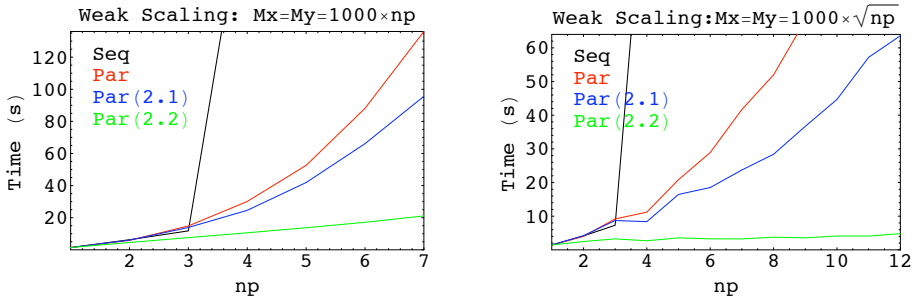

FIG. 6: Weak Scaling Results for the ScatterToZero, Parallel Partition (solve interface system on all processors), and optimized (v2.1, v2.2) version of the Parallel Partition method.

E. Version Workload Bar Charts

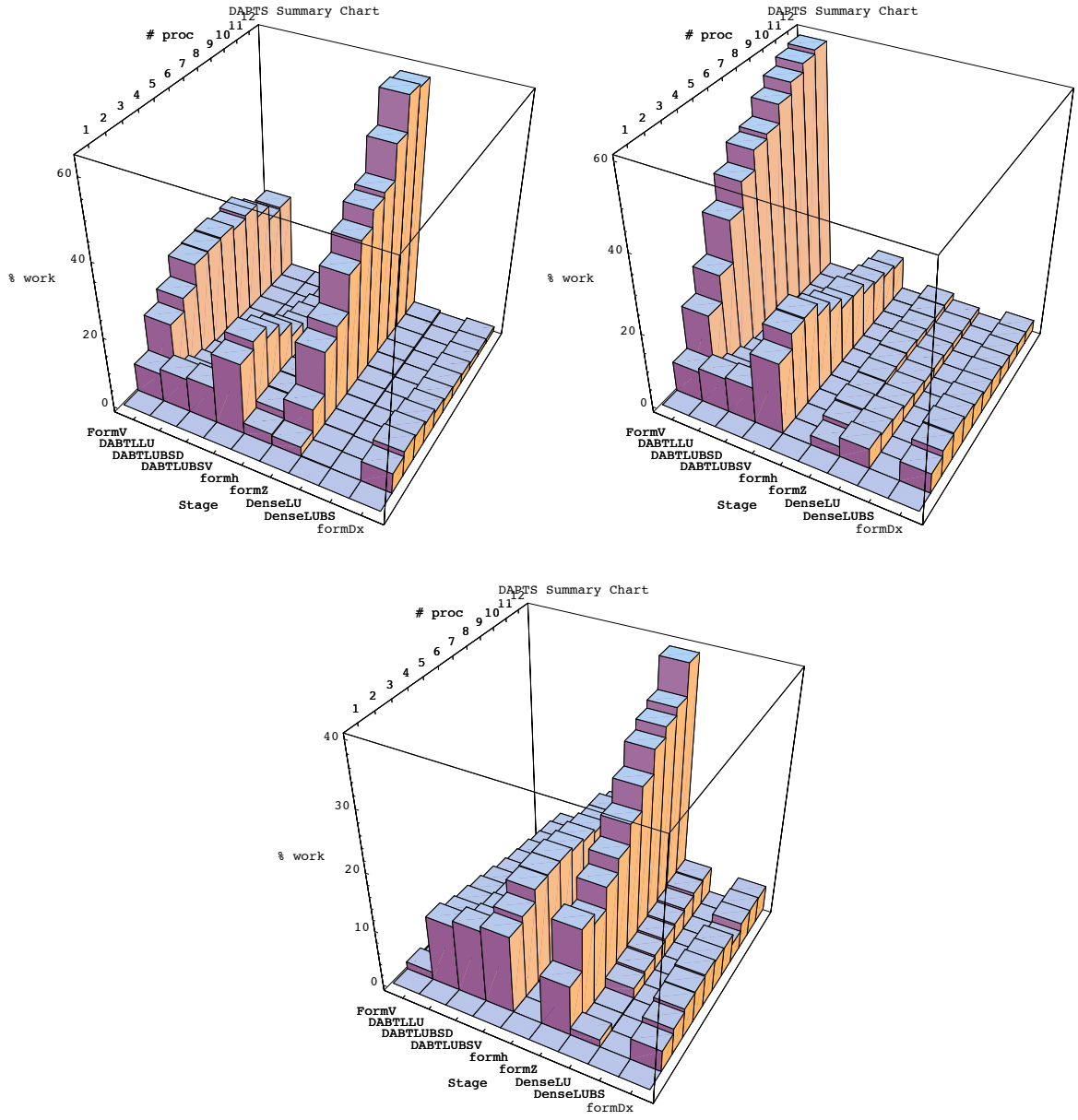

FIG. 7: (clockwise from top left) The communication of the $\mathcal{Z}$ kills the first version (v2.0) of the parallel solver. The optimized (v2.1) version, which is significantly faster, is slow in forming the $\mathcal{V}$ matrix. The second optimized version, is once again slow in forming $\mathcal{Z}$, although **significantly** faster than v2.0, v2.1

---

* Electronic mail: bag2107@columbia.edu; http://www.ap.columbia.edu/ctx/ctx.html

[1] Xian-He Sun, Hong Zhang, and Lionel M. Ni, *Efficient Tridiagonal Solvers on Multicomputers*, IEEE Transactions on Computers Vol. **41**, No. 3, March 1992.

[2] It should be stressed that the solution on a parallel computer should only be done by these methods if the Fast Poisson Solver is a module in a larger simulation, or the full system cannot fit on one processor (for the Partition Method).