

# HPSC Lab 9

Luna McBride

November 11 2023

## 1 Introduction

The non-linear methods presented in this lab provide an excellent starting point when attempting to manage non-linear systems. They come to the conclusions pretty swiftly given the immense complexity of a problem of that ilk. As powerful as they may be, however, they must be used intelligently with proper parallelization techniques to stand any chance of solving the massive problems high-performance scientific computing is known for. In order to use these methods intelligently though, a baseline understanding of their nuances must be established.

## 2 Algorithm and Test Case Development

### 2.1 Defining Terms

Jacobi: A mesh calculation method that uses ghost nodes and guessing to come toward a convergent value in a mesh system.

CG (Conjugate Gradient): A mesh calculation method that uses vector projections to come to a convergent value in a mesh system.

SA (Successive Approximation): Convert a non-linear problem into a linear one by guessing the values for the right hand side vector and using those to create a new guess, continuing until convergence.

NR (Newton-Raphson): A method for non-linear convergence that seeks to minimize the function  $f(\Phi) = Ax - b(\Phi)$  to where  $f(\Phi) = 0$ , using  $\Phi$  to turn the problem linear.

These acronyms will be used to represent the methods listed here moving forward.

### 2.2 Base Value Computations

#### 2.2.1 Initialized Values

The given code begins with a relaxation value of 0.3 ( $\text{relax} = 0.3$ ) with  $C_0$  and  $\tau$  (Tau) set to 1 ( $c_0 = 1.$ ,  $\text{tau} = 1.$ ).  $C_0$  and  $\tau$  are multipliers for the equation  $C_0 * e^{\tau\Phi}$ , which is used in the specified non-linearity equation where the second-derivative of  $f$  is equal to it. That is to say,  $\nabla^2\Phi$

$= C_0 * e^{\tau\Phi}$ . The relaxation parameter is then used to decide how much weight should be put into the new values. The exact update code with the relaxation parameter is as follows:

$$\text{phi}[r] = \text{relax} * \text{phi}[r] + (1 - \text{relax}) * \text{phiNew}[r] ;$$

This is updating the current phi to be the relaxation parameter times the current phi plus 1 minus the relaxation parameter times the new phi. Thus, the closer the relaxation parameter is to 1, the less weight is put toward the new guess.

### 2.2.2 Baseline Times

The following times were obtained using the timer functionality given in solvers.h. Start and end points were set before and after the qualifying functions, thus giving times accurate to the methods themselves rather than relying on the speed of the overall program. This was not enough in all cases, however, as both NR + Jacobi and SA + Jacobi could not finish within the one hour time allotted by Alpine’s acompile node. These cases are denoted by a time of N/A seconds to represent this.

The cell and PE counts may also look different in value, but they represent the same mesh sizes. For example, a one PE (1x1) mesh of 20x20 cells covers as many cells as a four PE (2x2) mesh of 10x10 cells, with each containing 400 cells. Therefore, the four columns represent two different meshes with 400 cells and 1600 cells respectively.

Method	1x1, nCell: 20x20	2x2, nCell: 10x10	1x1, nCell: 40x40	2x2, nCell: 20x20
Jacobi	0.125 sec	65.031 sec	1.727 sec	244.521 sec
CG	0.014 sec	3.747 sec	0.065 sec	7.888 sec
NR + Jacobi	1.372 sec	680.992 sec	17.513 sec	N/A sec
SA + Jacobi	1.013 sec	534.856 sec	14.124 sec	N/A sec
NR + CG	0.074 sec	29.064 sec	0.552 sec	61.509 sec
SA + CG	0.063 sec	21.781 sec	0.456 sec	45.595 sec

Table 1: CPU Run Times for Different Sized Meshes

The first aspect to note is the large drops in speed between the one PE (1x1) meshes and the four PE (2x2) meshes. These meshes may represent the same spaces physically, but each method takes at least one hundred times longer on the multi-PE meshes than on the single PE meshes. This means there is a lot of time leak coming from passing values between different PEs, which makes it a prime candidate for speed-up procedures.

Another important piece to note is how the Jacobi method leaks time compared to the CG method. All experiments showed between a ten and two hundred times increase in time, with all cases proving Jacobi to be the slower option. This is shown strongly in the case of NR, where CG spent 29 seconds compared to Jacobi’s 681 seconds (11.35 minutes). This comparison is also the one time in these cases where the wall times show a deeper story than otherwise given. In those same NR trials, CG had a wall time of 116.487 seconds (1.95 minutes) compared to Jacobi’s 2730.43 seconds (45.5 minutes).

All of this goes without mentioning the two Jacobi runs that could not finish within acompile’s

allocation time. For reference, it took the other non-linear models between 13 and 15 iterations to converge, even with the slower Jacobi times that did finish. The unfinished models got nowhere near that far in their execution. SA was stopped with three iterations, while NR was stopped at only one. Judging by these numbers and the average iterations before convergence, NR with Jacobi only made it somewhere between 1/15th and 1/13th of the way before it had to be stopped for running out the system’s one hour time frame. This shows that Jacobi forces exponentially more compute time when scaling the non-linear systems, thus making it unfit to use when testing their capabilities.

Test cases of the non-linear methods in the next section will be using CG due to this. The exact test case chosen will be specifically the four PE (2x2) mesh with 10x10 cells using CG. This may not be as fast as the single PE options, but it is suitably fast with enough time being taken to clearly show when a change in speed has occurred. It also utilizes multiple PEs, making it an excellent test case when updating MPI calls and general parallelization.

### 2.3 Changing Relaxation and Nonlinearity Parameters

These changes will be made individually, utilizing the default values for all values not being specifically altered. This is being done to prove that the hyperparameter change is causing the speed-up rather than any other controllable aspect.

Run	$C_0$	Tau	Relaxation	NR Time	SA Time
Baseline	1	1	0.3	29.064 sec	21.781 sec
$C_0$ Up	2	1	0.3	31.198 sec	22.955 sec
$C_0$ Down	0.5	1	0.3	27.968 sec	20.311 sec
Tau Up	1	2	0.3	32.250 sec	23.145 sec
Tau Down	1	0.5	0.3	27.824 sec	19.302 sec
Relaxation 0	1	1	0	29.181 sec	21.778 sec
Relaxation 0.5	1	1	0.5	29.484 sec	21.880 sec
Relaxation 1	1	1	1	29.328 sec	21.828 sec
Optimal	0.5	0.5	0.3	25.288 sec	17.957 sec

Table 2: Changes in Hyperparameters

The relaxation does not appear to do much within this context. All values remained almost the same for each non-linear method. As for  $C_0$  and Tau, both parameters sped up execution when their values went down while slowing down execution when their values went up. Therefore, lower values are better for those cases.

The more optimal values that will be used moving forward will be a relaxation of 0.3, a Tau of 0.5, and a  $C_0$  of 0.5 where applicable.

## 3 Suitability Results

CG may be faster than its cohort Jacobi, but there are still areas for improvement with it. Both non-linear methods are build by putting a loop around both CG and the method to form the mesh (FormLS). This makes these two functions, along with any functions they call, the prime suspects

for further speedup through parallelization. Being prime targets, however, does not necessarily make them guilty of the slowdown. This is where Intel Advisor comes in, as its powerful suitability tool will narrow down this search.

The suitability tool functions as a way to run what if scenarios on annotated parts of the code. These scenarios can show off how much extra load would be required to make different options viable. Threading provides the highest likelihood for speedup in this use case, so the function with the highest threading potential will be the best function for further focus of those previously identified targets.

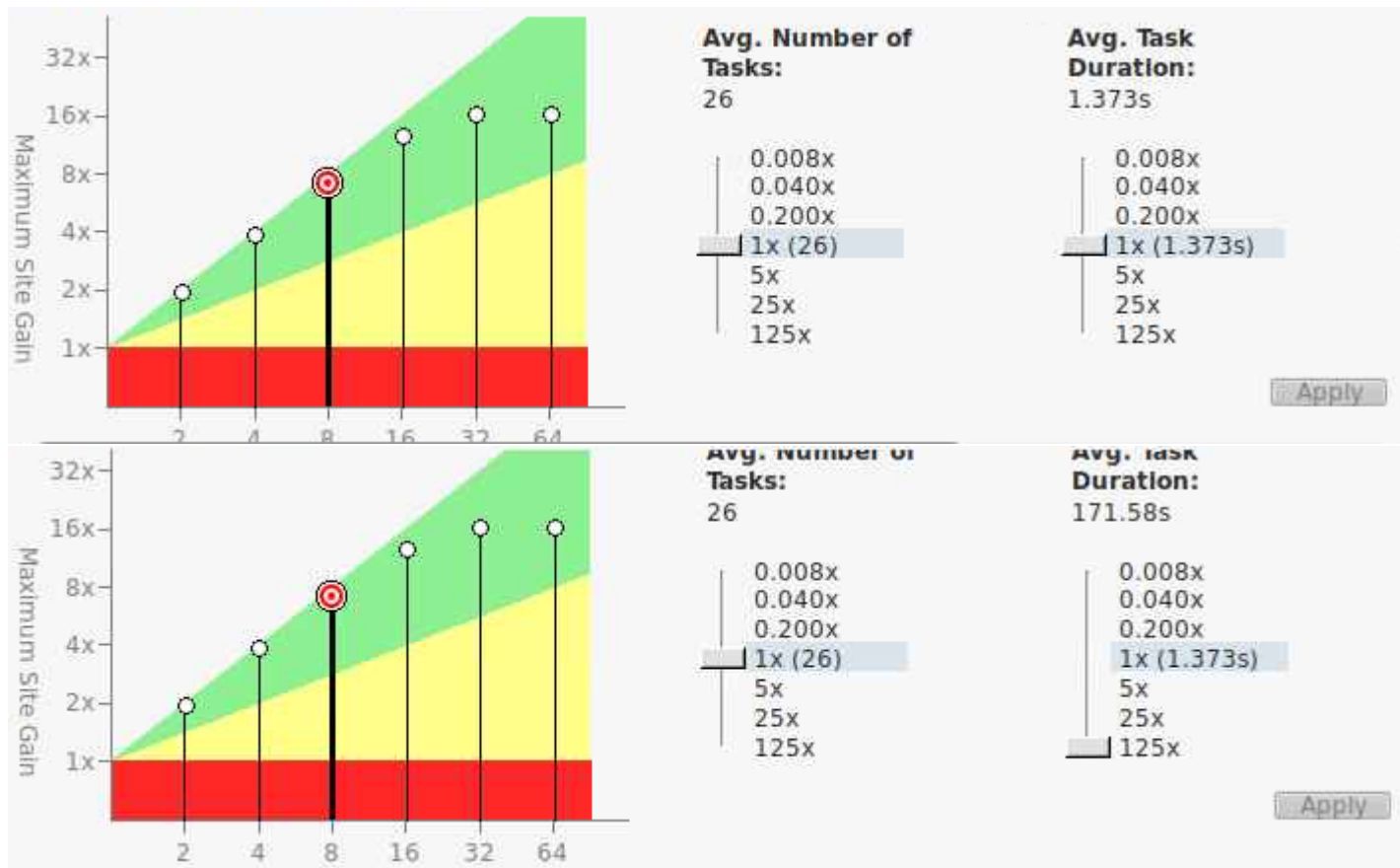
The annotation functions work by indicating different sites in the code, as well as certain tasks to focus on while running said site. The structure then shows off the potential of the site as a whole while showing the each separate loop or function as a task. This ability to denote tasks to view individually while still running the whole function as a site will prove useful in tracking both of these functions, as well as the non-linear loop itself. CG incorporates many individual functions to track while FormLS utilizes many different loops, making them the perfect sites for task tracking.

The following subsections will explore these areas with 3 images for each major section, representing a 20x20 mesh, a 200x200 mesh, as well as the 200x200 grid with Intel Advisor's duration setting set to 125x. This effectively builds a grid 125 times larger without having to run it. Intel Advisor, however, provides an error with the baseline values defined in the previous section, stating that the system runs too fast to analyze when running on a single PE for smaller grids. As such, all variables will need to be cranked up so that Intel Advisor can get the baseline of behavior in this case. The baseline values worked just fine when using both MPI and a 200x200 mesh, however, so these values were used in those cases.

It may seem easier to just scrap the 20x20 mesh case, but it represents a progression from previous labs. 20x20 was a giant, slow-moving behemoth in every lab until CG was introduced. It is a natural progression to see this oppressive force knocked down a peg in favor of a 200x200 grid being stretched 125 times by a powerful program like Intel Advisor.

### 3.1 Non-Linear Iterations





Source Code and Annotation Usage:

```

ANNOTATE_SITE_BEGIN(Nonlinear_iterations);
timingInfo nlTime(nlsolver + " with " + solver);
nlTime.Start(myMPI.myPE);
while ( global_converged == 0 && ++iter < max_iter )
{
    //          if ( myMPI.myPE == 0 ) { printf("\n\n");    printf("-- %s -- Iteration %d\n",nlsolver,iter); }

    ANNOTATE_TASK_BEGIN(Form);
    MESH.FormLS(myMPI,c0,tau);
    ANNOTATE_TASK_END();

    if ( nlsolver == "nr" )
    {
        if ( solver == "jacobi" ) {ANNOTATE_TASK_BEGIN(Jacobi);
            MESH.Jacobi(MESH.Jacobian, MESH.minusf, MESH.dPhi, myMPI ); ANNOTATE_TASK_END();}
        else if ( solver == "cg" ) {ANNOTATE_TASK_BEGIN(CG);
            MESH.CG (MESH.Jacobian, MESH.minusf, MESH.dPhi, myMPI );ANNOTATE_TASK_END();}
        else
            FatalError("Solver " + solver + " not found.");
    }
}

```

```

    converged = MESH.NR_Phi_Update( tol , relax );
}

    else if ( nlsolver == "sa" )
{
    if      ( solver == "jacobi" ) {ANNOTATE_TASK_BEGIN(Jacobi);
    MESH.Jacobi(MESH.Acoef , MESH.b , MESH.phiNew , myMPI ); ANNOTATE_TASK_BEGIN(Jacobi);}
    else if ( solver == "cg"      ) {ANNOTATE_TASK_BEGIN(CG);
    MESH.CG      (MESH.Acoef , MESH.b , MESH.phiNew , myMPI ); ANNOTATE_TASK_END();}
    else
        FatalError("Solver " + solver + " not found.");

    converged = MESH.SA_Phi_Update ( tol , relax );
}

    else
FatalError("Nonlinear Solver " + nlsolver + " not found.");

    MPI_Barrier(MPI_COMM_WORLD);    MPI_Allreduce(&converged, &global_converged, 1 , MPI_INT, MPI_MAX)

}

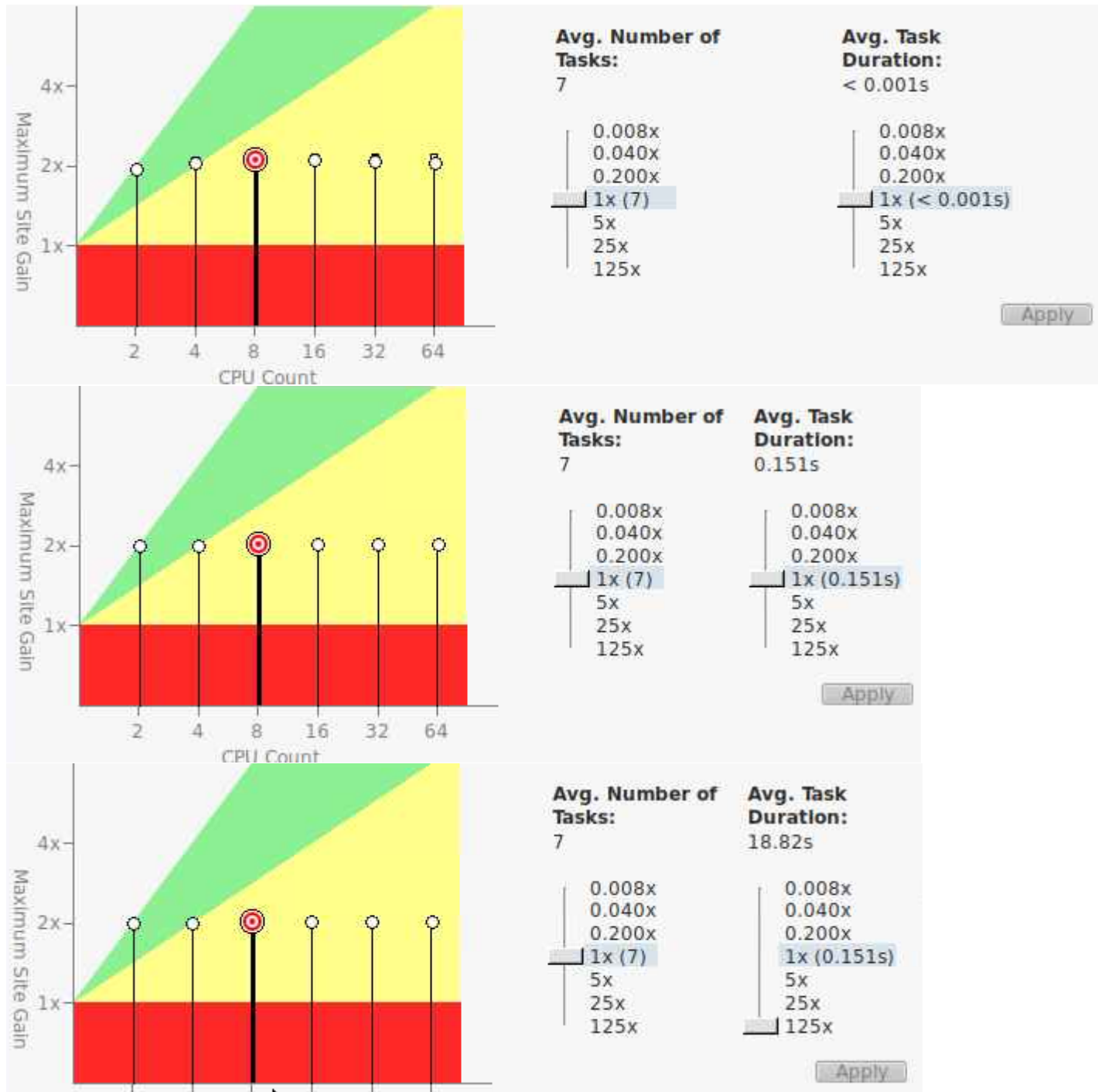
ANNOTATE_SITE_END();

if ( global_converged == 1 ) if ( myMPI.myPE == 0 ) cout << nlsolver << " converged in " << iterations << endl;
if ( global_converged == 0 ) if ( myMPI.myPE == 0 ) cout << nlsolver << " failed to converge after " << iterations << endl;

```

The most ideal graph for this analysis has all values on the diagonal edge of the green section. The Non-Linear section was meant to be here as a placeholder to show how the other functions beneath it would be the best targets to focus on, but these graphs clearly indicate an opposite situation. The values stay in the green even in the smaller mesh, making the main Non-Linearity loop a prime candidate for threading. It does not create that best case line along the diagonal like the most optimal case would, but it shows promise.

## 3.2 FormLS



Source Code and Annotation Usage:

```
void FormLS(mpiInfo &myMPI, double c0, double tau)
{
```

```

ANNOTATE_SITE_BEGIN(FormLS);
double half    = 0.5;
double quarter = 0.25;

ANNOTATE_TASK_BEGIN(Init);
rLOOP cLOOP Acoef  [r][c] = 0.; // Initialize linear system
rLOOP cLOOP Jacobian[r][c] = 0.; // Initialize Jacobian
rLOOP cLOOP Jcoef   [r][c] = 0; // The default is to point to entry zero; note that phi[0]
rLOOP      Jcoef    [r][1] = r; //
rLOOP      b        [r] = 0.;

double dx2 = dx*dx;          // Form matrix entries for the interior grid points
double dy2 = dy*dy;          // Form matrix entries for the interior grid points
ANNOTATE_TASK_END();
// Set up the E/W/N/S stencil:

ANNOTATE_TASK_BEGIN(Initialize_Form);
for ( int i = 1 ; i <= nRealx ; ++i )
    for ( int j = 1 ; j <= nRealy ; ++j )
{

    int p = pid(i,j);
        Jcoef[ p ][ 1 ] = p;
        if ( i < nRealx ) Jcoef[ p ][ 2 ] = p+1;          // East
    if ( i > 1 ) Jcoef[ p ][ 3 ] = p-1;          // West
    if ( j < nRealy ) Jcoef[ p ][ 4 ] = p+nRealx;      // North
    if ( j > 1 ) Jcoef[ p ][ 5 ] = p-nRealx;          // South

}

ANNOTATE_TASK_END();
// Loop over each cell, and let each cell contribute to its points' finite difference equation
ANNOTATE_TASK_BEGIN(Finite_Difference);
for ( int i = 1 ; i <= nCellx ; ++i )
    for ( int j = 1 ; j <= nCelly ; ++j )
{

    int p;

    p = pid(i,j);
    Acoef[p][1] += half    * ( -1./dx2 - 1./dy2 );
    Acoef[p][2] += half    *    1./dx2;
    Acoef[p][4] += half    *    1./dy2;
    b[p]        += quarter * c0*exp( tau * phi[p] );
}

```



```

p = pid(i+1,j);
Acoef[p][1] += half * ( -1./dx2 - 1./dy2 );
Acoef[p][3] += half * 1./dx2;
Acoef[p][4] += half * 1./dy2;
b[p] += quarter * c0*exp( tau * phi[p] );

p = pid(i+1,j+1);
Acoef[p][1] += half * ( -1./dx2 - 1./dy2 );
Acoef[p][3] += half * 1./dx2;
Acoef[p][5] += half * 1./dy2;
b[p] += quarter * c0*exp( tau * phi[p] );

p = pid(i,j+1);
Acoef[p][1] += half * ( -1./dx2 - 1./dy2 );
Acoef[p][2] += half * 1./dx2;
Acoef[p][5] += half * 1./dy2;
b[p] += quarter * c0*exp( tau * phi[p] );

}

ANNOTATE_TASK_END();
// Loop over each cell, and let each cell contribute to its points' Jacobian
ANNOTATE_TASK_BEGIN(Jacobian);
for ( int i = 1 ; i <= nCellx ; ++i )
    for ( int j = 1 ; j <= nCelly ; ++j )
{

int p;

p = pid(i,j);
Jacobian[p][1] += half * ( -1./dx2 - 1./dy2 ) - quarter * tau * c0 * exp( tau * phi[p] );
Jacobian[p][2] += half * 1./dx2;
Jacobian[p][4] += half * 1./dy2;

p = pid(i+1,j);
Jacobian[p][1] += half * ( -1./dx2 - 1./dy2 ) - quarter * tau * c0 * exp( tau * phi[p] );
Jacobian[p][3] += half * 1./dx2;
Jacobian[p][4] += half * 1./dy2;

p = pid(i+1,j+1);
Jacobian[p][1] += half * ( -1./dx2 - 1./dy2 ) - quarter * tau * c0 * exp( tau * phi[p] );
Jacobian[p][3] += half * 1./dx2;
Jacobian[p][5] += half * 1./dy2;

p = pid(i,j+1);

```

```

Jacobian[p][1] += half * ( -1./dx2 - 1./dy2 ) - quarter * tau* c0 * exp( tau * phi[p] );
Jacobian[p][2] += half * 1./dx2;
Jacobian[p][5] += half * 1./dy2;

    }
    ANNOTATE_TASK_END();
    // Apply BCs for SA
    ANNOTATE_TASK_BEGIN(SABC);
    for ( int i = 1 ; i <= nRealx ; ++i )
        for ( int j = 1 ; j <= nRealy ; ++j )
{

    int p = pid(i,j);
    if ( myMPI.iPE == 0 ) if ( i == 1 ) ApplyBC( Acoef , b, p, BCfunc(i,j) );
    if ( myMPI.iPE == myMPI.nPEx-1 ) if ( i == nRealx ) ApplyBC( Acoef , b, p, BCfunc(i,j) );
    if ( myMPI.jPE == 0 ) if ( j == 1 ) ApplyBC( Acoef , b, p, BCfunc(i,j) );
        if ( myMPI.jPE == myMPI.nPEy-1 ) if ( j == nRealy ) ApplyBC( Acoef , b, p, BCfunc(i,j) );

}

    ANNOTATE_TASK_END();

    // Compute right-hand side, f, for Newton system J*delta = -f. Note that f = A*phi - b

    ANNOTATE_TASK_BEGIN(minusf);
    rowLOOP
    {
    minusf[row] = b[row];
    colLOOP minusf[row] -= Acoef[row][col] * phi[ Jcoef[row][col] ] ;
    }
    ANNOTATE_TASK_END();

    // Apply BCs for NR
    ANNOTATE_TASK_BEGIN(NRBC);
    for ( int i = 1 ; i <= nRealx ; ++i )
        for ( int j = 1 ; j <= nRealy ; ++j )
{

    int p = pid(i,j);
    if ( myMPI.iPE == 0 ) if ( i == 1 ) {
        ApplyBC( Jacobian , minusf, p, phi[p] - BCfunc(i,j) ); }
    if ( myMPI.iPE == myMPI.nPEx-1 ) if ( i == nRealx ) {
        ApplyBC( Jacobian , minusf, p, phi[p] - BCfunc(i,j) ); }
    if ( myMPI.jPE == 0 ) if ( j == 1 ) {
        ApplyBC( Jacobian , minusf, p, phi[p] - BCfunc(i,j) ); }
        if ( myMPI.jPE == myMPI.nPEy-1 ) if ( j == nRealy ) {

```

```

        ApplyBC( Jacobian , minusf, p,  phi[p] - BCfunc(i,j) ); }

}

ANNOTATE_TASK_END();

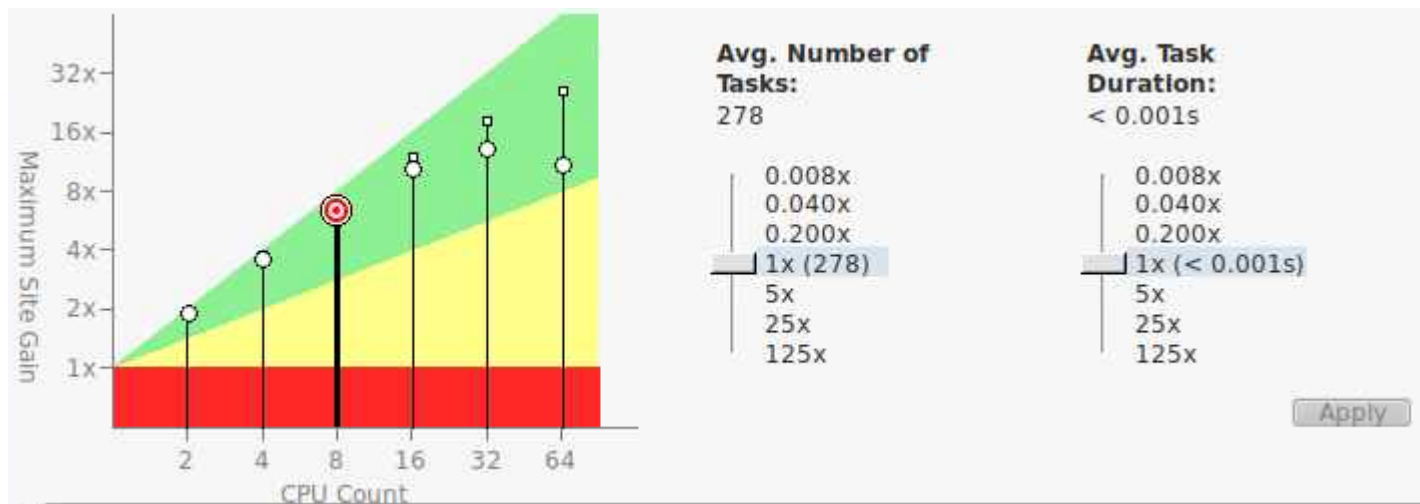
// debugging

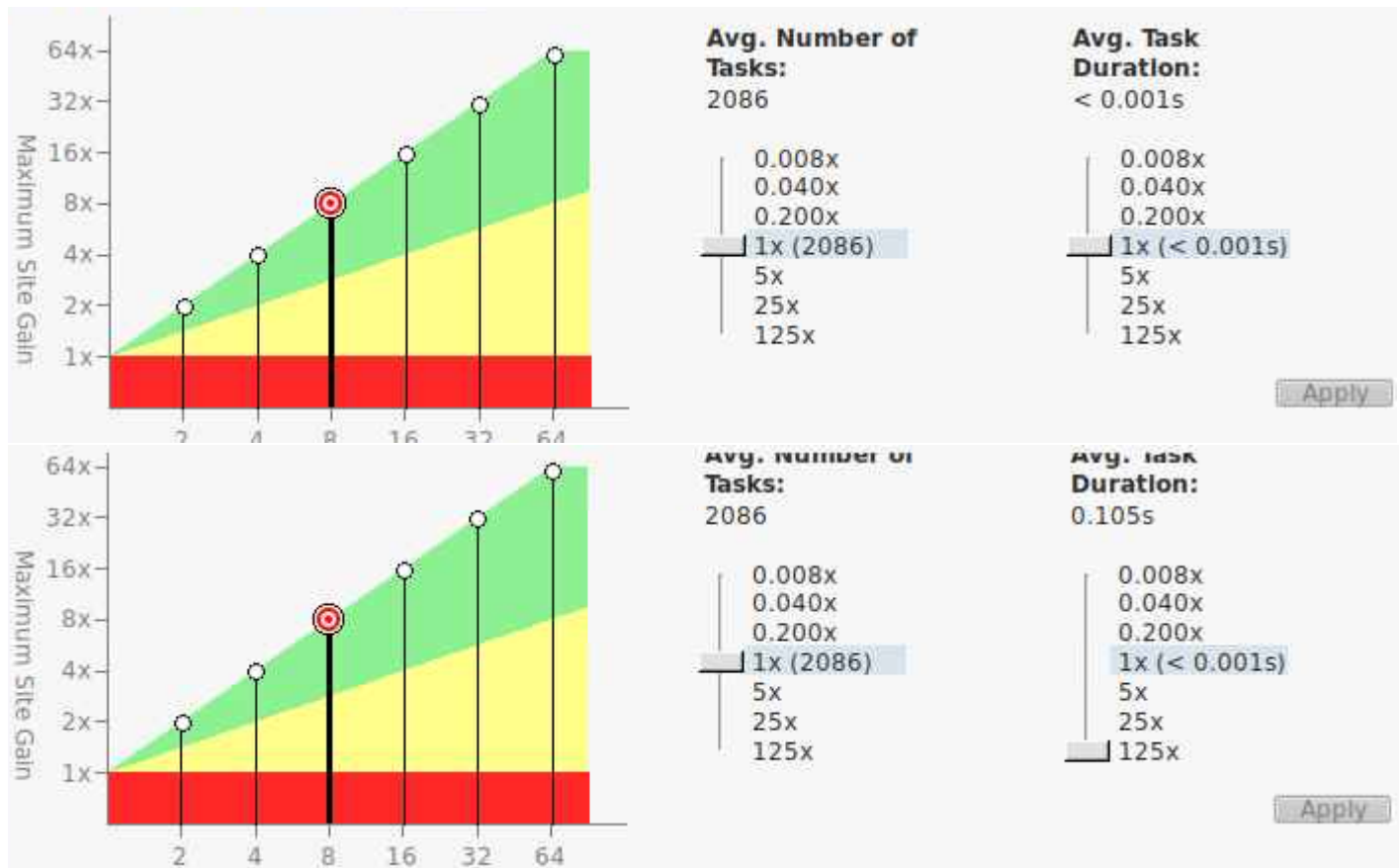
//    if ( printWhat == "printLinear") printLinearSystem();
//    if ( printWhat == "printNewton") printNewtonSystem();
ANNOTATE_SITE_END();
}

```

FormLS does not make for as good of a choice as its competitors. All grid size cases put it in the yellow for most thread sizes. This is still better than many of the cases, but threading alone will only do so much. This candidate may not be as strong in this regard, but the loops themselves are not very optimized to start with. FormLS is also set up in a way where it is building for both NR and SA cases every single time even if the other cases are not being used. Therefore, combination of loop fixes and threading could make it stand toe to toe with the others. It just does not hold that same inherent promise that the other sites hold.

### 3.3 CG





Source Code and Annotation Usage:

```
void CG(VDD &Matrix , VD &RHS , VD &Solution , mpiInfo & myMPI)
{
    VD rnew; rnew.resize(nField + 1);
    VD r;      r.resize(nField + 1);
    VD p;      p.resize(nField + 1);
    VD Ap;     Ap.resize(nField + 1);
    VD Ax;     Ax.resize(nField + 1);

    double p_dot_Ap;          // Stores matrix-vector product
    double r_dot_r;           // Stores r dot r
    double rnew_dot_rnew;     // Stores rnew dot rnew
    double alpha;             // Alpha in the above-referenced algorithm
    double beta;              // Beta " " " " " "
    double cur_delta;
    double tol                = 1.e-15;
    int global_converged = 0;
    int iter                = 0;
    int max_iter            = nField * 1000;
}
```

```

int converged;

// (1) Initial guess and other initializations

rowLOOP p[row] = r[row] = 0.;

Solution[0] = 0.;
p          [0] = 0.;
r          [0] = 0.;

// (2) Prepare for parallel computations on RHS

VD b_PESum ;
b_PESum.resize(nField + 1 ) ;
rowLOOP b_PESum[row] = RHS[row];
myMPI.PESum(b_PESum);

// (3) Initialize residual, r, and r dot r for CG algorithm

Residual(Matrix , r , Solution,b_PESum,myMPI);

rowLOOP p[row] = r[row];

r_dot_r = Dot(r,r,myMPI);

// (4) CG Iterations

ANNOTATE_SITE_BEGIN(CG_iterations);
while ( global_converged == 0  && ++iter <= max_iter)
{
    // (4.1) Compute alpha

    ANNOTATE_TASK_BEGIN(MatVecProd);
    MatVecProd(Matrix,p,Ap,myMPI);      // A*p (stored in Ap)
    ANNOTATE_TASK_END();

    ANNOTATE_TASK_BEGIN(Dot);
    p_dot_Ap = Dot(p,Ap,myMPI); // p*Ap
    ANNOTATE_TASK_END();

    alpha     = r_dot_r / p_dot_Ap;

    // (4.2) Update solution and residual
    ANNOTATE_TASK_BEGIN(R_Update);

```

```

rowLOOP Solution[row] = Solution[row] + alpha * p[row];
rowLOOP rnew      [row] = r[row]          - alpha * Ap[row];

// (4.3) Compute beta

rnew_dot_rnew = Dot(rnew,rnew,myMPI);
beta          = rnew_dot_rnew / r_dot_r;

// (4.4) Update search direction

rowLOOP p[row] = rnew[row] + beta*p[row];

// (4.5) r "new" will be r "old" for next iteration

rowLOOP r[row] = rnew[row];
r_dot_r      = rnew_dot_rnew;
ANNOTATE_TASK_END();

// (4.6) Check convergence on this PE

ANNOTATE_TASK_BEGIN(Check_Converge);
rowLOOP
{
cur_delta = fabs(alpha * p[row]);
if ( cur_delta > tol ) converged = 0;
}

    if ( fabs(r_dot_r) < tol)
converged = 1;
    else
converged = 0;
    ANNOTATE_TASK_END();

// (4.7) Check convergence across PEs, store result in "global_converged"

MPI_Allreduce(&converged, &global_converged, 1 , MPI_INT, MPI_MIN, MPI_COMM_WORLD);
}
ANNOTATE_SITE_END();

// (5) Done - Inform user

if ( global_converged == 1 ) if ( myMPI.myPE == 0 ) cout << " (o) CG converged in " << iter << endl;
if ( global_converged == 0 ) if ( myMPI.myPE == 0 ) cout << " (o) CG failed to converge after " << iter << endl;

```

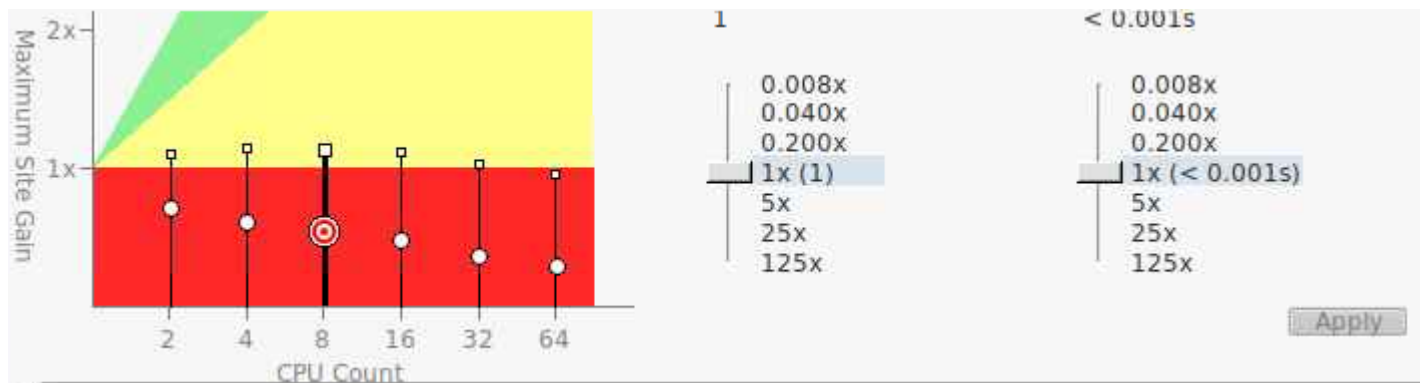
}

CG shows high gain at even the lowest of sizes. It remains in the green as an interesting site to target with the values creating the expected line in both size-increased cases. It may not show that trend entirely in the very small first case, but it is clearly approaching that conclusion even at such a small size. This would make an excellent site for threading.

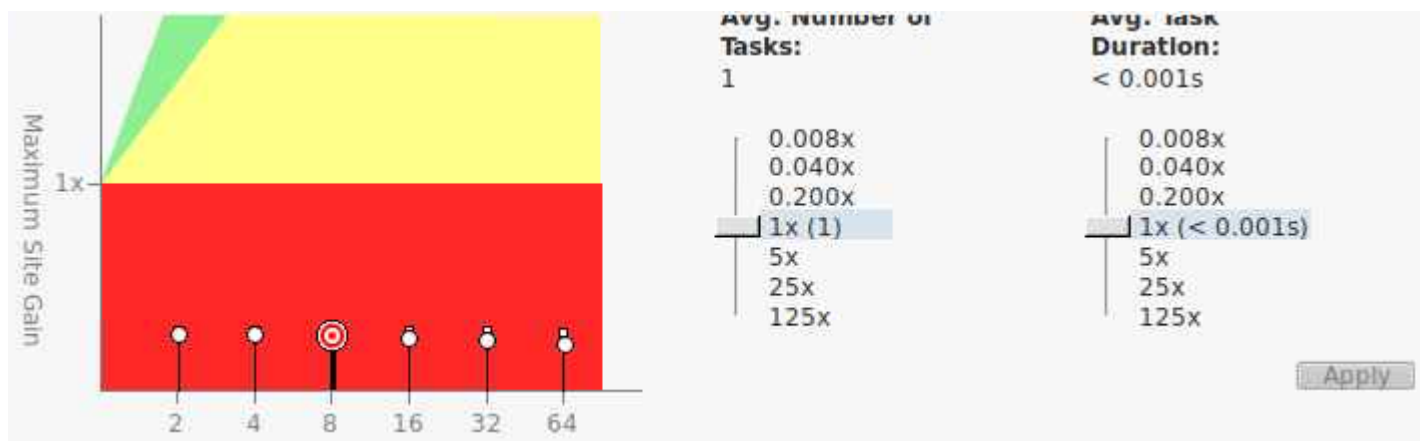
### 3.4 Honorable Mentions

This section is for the cases that did not make sense to look into deeper due to poor scores in the 1x section. These variables may very well prove more promising at larger values. The previous sites were explored further due to their promise displayed even before increasing sizes.

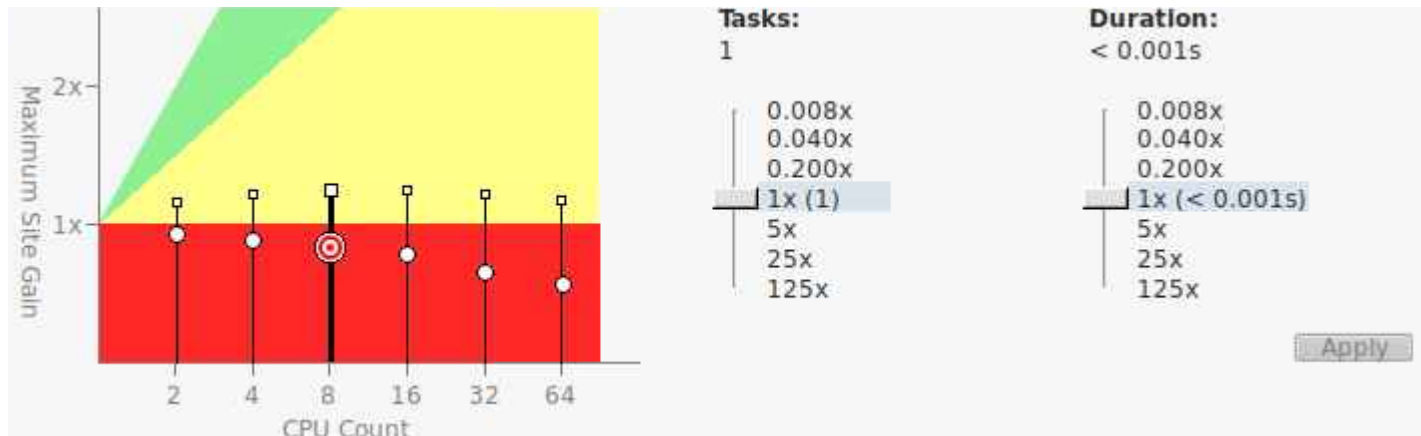
setBC:



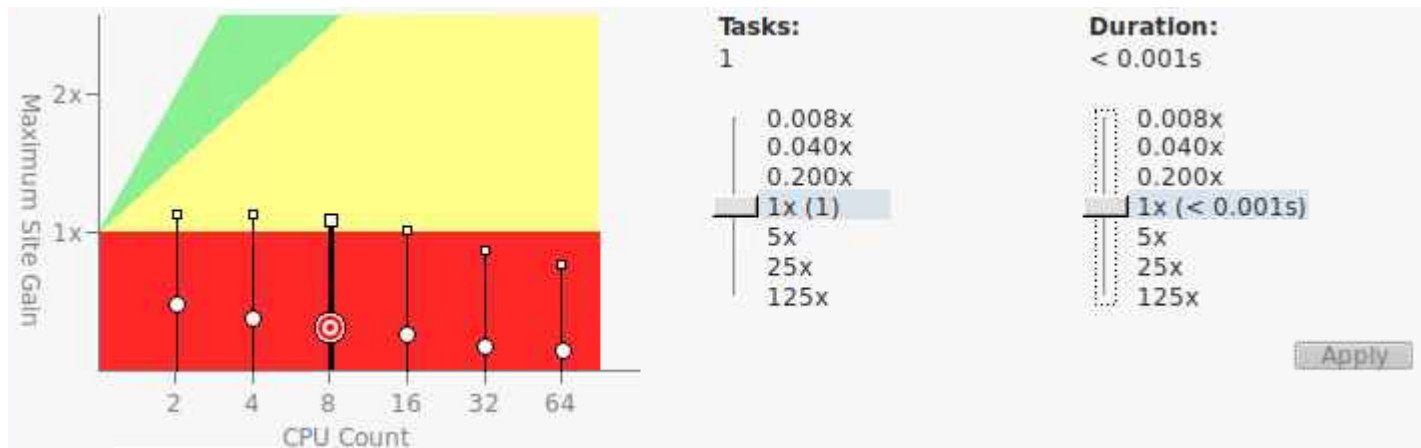
Residual:



MatVecProd:



Dot:



Source Code and Annotation Usage:

```
void ApplyBC(VDD &Matrix , VD &RHS , int BCrow,double BCvalue)
{
```

```
    cLOOP Matrix[BCrow][c] = 0.;
    cLOOP Jcoef [BCrow][c] = 0 ;
```

```
    Matrix[ BCrow ] [ 1 ] = 1.      ;
    Jcoef [ BCrow ] [ 1 ] = BCrow   ;
    RHS   [ BCrow ]      = BCvalue ;
```

```
    ANNOTATE_SITE_BEGIN(setBC);
    ANNOTATE_ITERATION_TASK(BCrowLoop);
```

```
    rLOOP
        if ( r != BCrow )
```

```
{
```



```

cLOOP
{
    if ( Jcoef[r][c] == BCrow )
{
    RHS[r] -= Matrix[r][c]*BCvalue;
    Jcoef[r][c] = 0; Matrix[r][c] = 0.;
}
}
}
    ANNOTATE_SITE_END();
}

double Dot(VD &vec1, VD &vec2 , mpiInfo &myMPI)
{
    ANNOTATE_SITE_BEGIN(Dot);
    double sum = 0.;

    // peMulticity = 1 except on PE boundaries where it is equal to the
    // number of PEs containing the given node.
    ANNOTATE_TASK_BEGIN(Sum);
    rLOOP sum += vec1[r]*vec2[r] / myMPI.peMultiplicity[r];
    ANNOTATE_TASK_END();
    // Sum results across PEs and share that sum with all PEs

    double globalSum;
    MPI_Allreduce(&sum, &globalSum, 1 , MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    ANNOTATE_SITE_END();
    return globalSum;
}

void MatVecProd(VDD &Matrix , VD &p , VD &prod , mpiInfo &myMPI)
{
    // Serial computation on this PE
    ANNOTATE_SITE_BEGIN(MatVecProd);
    ANNOTATE_TASK_BEGIN(MainLoop);
    rowLOOP
    {
        prod[row] = 0.;
        colLOOP
    {
        int Acol = Jcoef[row][col];
        if ( Acol > 0 ) prod[row] += Matrix[row][col] * p[ Acol ];
    }
}
}

```

```

    }
    ANNOTATE_TASK_END();
    // Handle PE boundaries

    myMPI.PEsum(prod);
    ANNOTATE_SITE_END();
}

void Residual(VDD &Matrix , VD &residual , VD &Sol , VD &RHS, mpiInfo &myMPI)
{
    ANNOTATE_SITE_BEGIN(Residual);
    MatVecProd(Matrix, Sol , residual , myMPI);



    ANNOTATE_TASK_BEGIN(rowLOOP);
    rowLOOP residual[row] = RHS[row] - residual[row];
    ANNOTATE_TASK_END();
    ANNOTATE_SITE_END();
}

```

As can be seen here, all of these at the smallest size provided net negative values at the smallest size. This, once again, does not necessarily mean these are bad choices to focus on. It just means that they would need to be scaled higher to show that promise. Compare that to CG, FormLS, and even the Non-Linear function itself, where gain was either already in the green for smaller grid sizes or, in the case of FormLS, had extra points reaching toward green to show its possible speedup. All three of those could show potential speedup even with smaller sizes of meshes, thus making them easier to test parallel speedup without needing to rely on larger time frames to run them. It is still best to test the threaded solutions at bigger sizes, but the changes from these functions will be a lot more subtle compared to the others functions.

## 4 Threading Results Comparison

This may seem shocking, but the first place to look would likely be the Non-Linear starting loop. It may not have created that perfect line like CG or approached it like FormLS, but that factor matters less compared to approximate impact time, as shown below:

Site Label	Source Location	Impact to Program Gain▼	Combined Site Metrics, All Instance	
			Total Serial Time	Total Parallel T
Nonlinear_iterations	 solvers.cpp:164	7.07x	555680.33s	78147.75s
CG_iterations	 llinearSolver.h:240	2.17x	342207.27s	42863.67s
FormLS	 LaplacianOnGrid.h:134	1.51x	213932.52s	26893.34s

This is the approximate impact calculated in Intel Advisor. These values shift around as other values have their duration changed, so it is important to note that this case has all three of those

important functions set to 125 times duration. This means the actual loop behind the Non-Linearity holds the key to significantly more impact from threading than its internal functions. To make it more clear, here is this same impact table with Non-Linearity's duration set to 1 time while both FormLS and CG maintain a duration value of 125 times:

Site Label	Source Location	Impact to Program Gain▼	Combined Site Metrics, All Instances	
			Total Serial Time	Total Parallel Time
CG_iterations	linearSolver.h:240	7.63x	3.17e+07s	3972591.16s
Nonlinear_iterations	solvers.cpp:164	6.93x	3.19e+07s	4576370.90s
FormLS	LaplacianOnGrid.h:134	1.01x	210545.26s	26442.27s
setBC	LaplacianOnGrid.h:280	1.00x	210514.46s	210514.67s

CG may have jumped up tremendously to surpass Non-Linearity, but the Non-Linearity will increase impact almost exactly as much whether it is running at 1 time or 125 times duration. This resulting table may not represent the reality of the problem at hand, but it shows just how powerful threading on the main loop would be.

Another aspect to note is the individual iterations that must occur in both the Non-Linear solver and CG. As previously noted, the Non-Linear solver is expected to undergo thirteen to fifteen iterations to converge. This does not change in the case of NR in a 200x200 mesh, as shown from the output of the run listed below:

```
[timingInfo] [nr with cg] Start time (Wall): 516622 Start time (CPU) : 6996769
(o) CG converged in 642 iterations.
(o) CG converged in 635 iterations.
(o) CG converged in 615 iterations.
(o) CG converged in 566 iterations.
(o) CG converged in 543 iterations.
(o) CG converged in 519 iterations.
(o) CG converged in 497 iterations.
(o) CG converged in 475 iterations.
(o) CG converged in 453 iterations.
(o) CG converged in 429 iterations.
(o) CG converged in 404 iterations.
(o) CG converged in 373 iterations.
(o) CG converged in 348 iterations.
nr converged in 13 iterations.
```

This is very important to note when discussing threading, as a barrier needs to be placed before the loop's end to ensure one thread does not rush past the others. Thirteen iterations means thirteen barriers that OMP will need to wait on. Compare that to the CG algorithm that, in this use case, would need 6,499 barriers for the amount of iterations it runs. This amount would only get larger as the problem scales.

There is also only one major difference between these two sites, being how many iterations of CG are accounted for. Threading would necessarily need to occur starting after FormLS in the Non-Linear loop to ensure the update to the mesh, leaving only CG and convergence checks for the Non-Linear loop to split. Convergence checks notwithstanding (given that threaded applications also need to keep up with convergence), CG is now the only portion being threaded in both cases. The question then becomes if threading the CG function as a whole like happens in the Non-Linear loop would make more of an impact than threading a single iteration, which is what would happen when threading CG.

This results in two factors that must be used to make the decision, being CG iterations and barriers. The barriers, as previously discussed, will occur at the end of each threaded iteration. The Non-Linear solver may have less iterations, but this barrier may be a larger hurdle given that it is waiting on the entire CG algorithm rather than one iteration. The difference between thirteen potentially larger barriers and thousands of smaller barriers is hard to compare without running both cases, so this factor may become very big or very small depending on the mesh size.

The bigger factor actually becomes the number of CG iterations. CG is going to run the same number of iterations no matter if these iterations are threaded or not. Threading a single iteration only splits the specified vectors up for calculation rather than any factors going into convergence of the algorithm, meaning that the number of iterations is an aspect independent from the actual threading. Compare this notion to the Non-Linear loop threading on the whole function, which reduces the size of the mesh CG is being called on. For the sake of example, another round of NR has been run on a mesh size of 200x50. This is the same size of mesh the Non-Linear loop would call on each thread of a 4-threaded application. This run is shown below:

```
[timingInfo] [nr with cg] Start time (Wall): 531866 Start time (CPU) : 1506436
(o) CG converged in 476 iterations.
(o) CG converged in 471 iterations.
(o) CG converged in 457 iterations.
(o) CG converged in 445 iterations.
(o) CG converged in 431 iterations.
(o) CG converged in 414 iterations.
(o) CG converged in 395 iterations.
(o) CG converged in 382 iterations.
(o) CG converged in 361 iterations.
(o) CG converged in 328 iterations.
(o) CG converged in 306 iterations.
(o) CG converged in 285 iterations.
(o) CG converged in 264 iterations.
(o) CG converged in 246 iterations.
nr converged in 14 iterations.
```

This run completed 5261 CG iterations, which is 1238 CG iterations fewer than the 6499 CG iterations completed during the 200x200 mesh run. That means that the four threads running in parallel would need to complete around twenty percent fewer CG iterations on average to complete

a single NR run. This factor alone makes the Non-Linear loop a much better candidate to look into than CG. It also allows for SA and NR to be compared on a threading level on top of previous comparisons in the lab. These are just based on theoretical results, however. The actual results hold more merit than any idea that can be posited at this stage.

Jacobi was ignored in this implementation and comparison for obvious reasons. Jacobi could not hold its own against the 20x20 mesh, so there is no way it could stand against the 200x200 meshes it would need to in order to have any leg in this race. It is also important to note that the implementation of non-linear threading can draw merits from altering FormLS and CG, just in case any of the intuition above turns out to be incorrect.

This did end up being the case. The hope was to have all of the threads simply split the data following the intuition from Lab 7, replacing the typical rowLOOPS with a modified vecLOOP by adding this code to solvers.h: `#define vecLOOP for (int row = Lower; row != Upper; ++row)`. This loop would then build the full mesh piece-wise, resulting in one proper entity build by multiple separate parts. FormLS took to this intuition well, building its mesh exactly as expected with following the proper tweaks. CG, however, did not.

CG would often create little kinks in the mesh when using vecLOOP. The kinks may have been slight on a 20x20 mesh, but they slowed down convergence on larger meshes. Following this, attempts were made to send split up versions of the vectors to CG while utilizing a further modified sizeLOOP, defined in solvers.h as `#define sizeLOOP for (int row = 1; row != size; ++row)`. Size was calculated as the interval between Lower and Upper. This proved more fruitful, but it did not thread as expected. Splitting vectors and arrays in C++ is also very costly, as the compiler also would only take such splits if the vectors were inserted into another vector. The 200x200 mesh slowed down significantly due to this. Python would be better for this intuition for this reason. As such, the major time save came down to FormLS in this case.

Below are the final results for the threading. Included are the non-threaded Linear Solver times, as some of the changes to enable threading created speedup in the base case:

Method	1x1, nCell: 20x20	1x1, nCell: 200x200	1x1, nCell: 500x500
CG Old	0.011 sec	6.219 sec	97.330 sec
CG New	0.0097 sec	5.540 sec	89.015 sec
SA Old	0.065 sec	51.507 sec	754.485 sec
SA New	0.110 sec	32.112 sec	487.35 sec

Table 3: Threading Speedup, Number of Threads = 4

FormLS may have been the main factor threaded, but it still brought the 500x500 grid's time down by over 35 percent. The 20x20 mesh may have gone up a little bit, but this small increase is a minor price to pay for the major speedup for much larger grids.

## 5 Thread/MPI Rank Trade-off

Method	1x1, nCell: 200x200	2x1, nCell: 200x200	2x2, nCell: 200x200
CG 4 Threads	5.540 sec	61.536 sec	120.017 sec
CG 8 Threads	5.139 sec	61.600 sec	120.213 sec
SA 4 Threads	32.112 sec	356.546 sec	645.442 sec
SA 8 Threads	30.084 sec	357.471 sec	652.873 sec

Table 4: MPI + Threading Comparison

The increased threads did not help. Intel Advisor had already shown such for FormLS fixes, so this was to be expected. The initial intended routes of fixing CG as well likely would have likely given some better results based on Intel Advisor’s visualizations, but so goes the coding lifestyle.

As for MPI, the changes made made it slower on average. This is not a change specific to threading. A 2x2 PE run done prior to threading had the linear solver finish in 82.947 seconds, a far cry from the 120.017 seconds that it took to run after changes were made. It is also important to highlight that these times come from the linear solver alone, meaning that the changes to allow for the threading made the system harder to parallelize. This is a shocking aspect to note, as turning nested for loops into a single for loop, like was done for FormLS, usually speeds up a program rather than slow it down. There is clearly more to lining up these two powerful tools than meets the eye. It really just comes down to how one threads the needle.

It is important not to forget that MPI can hold a system back if not done well regardless. Intel Advisor had MPI functions take up three of the five biggest spots for slowdown when it was tested with MPI-run code. Such values were not used for analysis in the suitability section for that reason, as the majority of people experimenting with this style of code will not have access to the underlying code in order to change it.

## 6 Self Evaluation

This lab proved to be very difficult. Days worth of time went toward trying to get the CG threaded, only to have to throw in the towel as the deadline looms. Of course the deadline is at 1:25pm, but it is time to sleep in after days of only doing homework. More than just threading was done for speedup as well, mostly as FormLS needed to be converted from i,j formatted loops to pid formatted loops to ensure a more even split. This also ended in some extra time save, as nested loops and their  $O(N^2)$  is always going to be worse than singular loops at  $O(N)$ . There was also some good time save on shrinking various vectors down when attempting to implement sizeLOOP, but it was not enough to beat out the vector definitions. Python is far superior in this regard, with syntax of `array[n:m]` for an array from n to m that can just be passed into the function instead of having to be put into a variable before in can be sent to the function. Being used to such an easy to code and vectorize solution like this did not do any favors when it came to trying to come up with solutions to this problem. Perhaps someone with deeper c++ knowledge could do it.

One more aspect to note in regard to the slowdown is the limits Alpine has when it comes to job submissions. Attempts were made to allow for a larger time limit for the 8 thread 2x2 PE

200x200 mesh case, but Alpine limits such requests to an hour time. Below is the error that came from these attempts. This is being emphasized not just due to this use case, but also as a point of comparison for the 20x20 mesh sizes of Jacobi from the initial exploration. CG can run a 200x200 mesh system while being bogged down by MPI slowdown in around an hour, while Jacobi could not handle a 20x20 mesh in that same timeframe.

sbatch: error: QOSMaxWallDurationPerJobLimit

sbatch: error: Batch job submission failed: Job violates accounting/QOS policy (job submit limit, user's size and/or time limits)

## 7 Appendix A

Nonlinear Code:

```
// Nonlinear iterations
```

```
ANNOTATE_SITE_BEGIN(Nonlinear_iterations);
timingInfo nlTime(nlsolver + " with " + solver);
nlTime.Start(myMPI.myPE);
```

```
while ( global_converged == 0 && ++iter < max_iter )
{
```

```
    //          if ( myMPI.myPE == 0 ) { printf("\n\n");    printf("-- %s -- Iteration %d\n",nlsolv
```

```
    #pragma omp parallel
    {
        int myTH = omp_get_thread_num();
```

```
        numPerTh = nField / numTH;
        int Lower = (numPerTh * myTH) + 1;
        int Upper;
```

```
        if(myTH == numTH - 1) Upper = nField;
        else Upper = Lower + numPerTh - 1;
```

```
        int size = Upper - Lower;
        ANNOTATE_TASK_BEGIN(Form);
        MESH.FormLS(myMPI,c0,tau, Lower, Upper, nlsolver);
        ANNOTATE_TASK_END()
```

```
    #pragma omp barrier
```

```

}
    if ( nlsolver == "nr" )
{
    if      ( solver == "jacobi" ) {ANNOTATE_TASK_BEGIN(Jacobi);
        MESH.Jacobi(MESH.Jacobian, MESH.minusf, MESH.dPhi, myMPI ); ANNOTATE_TASK_END();}
    //else if ( solver == "cg"      ) {ANNOTATE_TASK_BEGIN(CG);
        MESH.CG      (MESH.Jacobian, MESH.minusf, MESH.dPhi, myMPI );ANNOTATE_TASK_END();}
    else if ( solver == "cg"      ) {ANNOTATE_TASK_BEGIN(CG);
        MESH.CG      (MESH.Jacobian, MESH.minusf, MESH.dPhi, myMPI, 1, nField, nField);ANNOTATE_TASK_END();}
    else
        FatalError("Solver " + solver + " not found.");

    converged = MESH.NR_Phi_Update( tol , relax );
}

    else if ( nlsolver == "sa" )
{
    //VDD aCo(MESH.Acoef.begin()+Lower, MESH.Acoef.begin()+Upper+1);
    //VD bCo(MESH.b.begin()+Lower, MESH.b.begin()+Upper+1);
    //VD ph(MESH.phiNew.begin()+Lower, MESH.phiNew.begin()+Upper+1);

    //#pragma omp barrier

    if      ( solver == "jacobi" ) {ANNOTATE_TASK_BEGIN(Jacobi);
        MESH.Jacobi(MESH.Acoef , MESH.b , MESH.phiNew , myMPI ); ANNOTATE_TASK_BEGIN(Jacobi);}
    else if ( solver == "cg"      ) {ANNOTATE_TASK_BEGIN(CG);
        MESH.CG      (MESH.Acoef , MESH.b , MESH.phiNew , myMPI, 1, nField, nField ); ANNOTATE_TASK_END();}
    // else if ( solver == "cg"      ) {ANNOTATE_TASK_BEGIN(CG);
        MESH.CG      (aCo, bCo, ph, myMPI, Lower, Upper, size ); ANNOTATE_TASK_END();}
    else
        FatalError("Solver " + solver + " not found.");
    // #pragma omp barrier
    converged = MESH.SA_Phi_Update ( tol , relax );
}

    else
FatalError("Nonlinear Solver " + nlsolver + " not found.");

    MPI_Barrier(MPI_COMM_WORLD);    MPI_Allreduce(&converged, &global_converged, 1 , MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    //}
}

```



FormLS and ApplyBC (Only difference in ApplyBC being rowLOOP becoming vecLOOP)

```

    // ==
//  ||
//  ||  Form Linear System Ax = b
//  ||
//  ==

void FormLS(mpiInfo &myMPI, double c0, double tau, int Lower, int Upper, string solver)
{
    ANNOTATE_SITE_BEGIN(FormLS);

    ANNOTATE_TASK_BEGIN(Init);
    vecLOOP cLOOP Acoef    [row][c] = 0.; // Initialize linear system
    vecLOOP cLOOP Jacobian[row][c] = 0.; // Initialize Jacobian
    vecLOOP cLOOP Jcoef     [row][c] = 0 ; // The default is to point to entry zero; note that phi
    vecLOOP      Jcoef     [row][1] = row; //
    vecLOOP      b         [row] = 0.;

    double dx2 = dx*dx;          // Form matrix entries for the interior grid points
    double dy2 = dy*dy;          // Form matrix entries for the interior grid points

    double half    = 0.5;
    double quarter = 0.25;
    ANNOTATE_TASK_END();
    // Set up the E/W/N/S stencil:
    int p;
    int i;
    int j;

    ANNOTATE_TASK_BEGIN(Initialize_Form);
    for (int pt = Lower ; pt <= Upper ; ++pt )
{
    p = pt;
    j = (p-1) / nRealy ;
    i = (p-1) - j*nRealx; j+=1; i+=1;

        Jcoef[ p ][ 1 ] = p;
        if ( i < nRealx ) Jcoef[ p ][ 2 ] = p+1;          // East
    if ( i > 1          ) Jcoef[ p ][ 3 ] = p-1;          // West
    if ( j < nRealy    ) Jcoef[ p ][ 4 ] = p+nRealx;      // North
    if ( j > 1          ) Jcoef[ p ][ 5 ] = p-nRealx;      // South
}
}

```

```

    ANNOTATE_TASK_END();
    // Loop over each cell, and let each cell contribute to its points' finite difference equation
    ANNOTATE_TASK_BEGIN(Finite_Difference);
    #pragma omp barrier
    for (int pt = Lower ; pt <= Upper ; ++pt )
{
    p = pt;
    j = (p-1) / nRealy ;
    i = (p-1) - j*nRealx; j+=1; i+=1;

    if(i==nRealx or j==nRealy) continue;

    Acoef[p][1] += half * ( -1./dx2 - 1./dy2 );
    Acoef[p][2] += half * 1./dx2;
    Acoef[p][4] += half * 1./dy2;
    b[p] += quarter * c0*exp( tau * phi[p] );

    p = pid(i+1,j);
    Acoef[p][1] += half * ( -1./dx2 - 1./dy2 );
    Acoef[p][3] += half * 1./dx2;
    Acoef[p][4] += half * 1./dy2;
    b[p] += quarter * c0*exp( tau * phi[p] );

    p = pid(i+1, j+1);
    Acoef[p][1] += half * ( -1./dx2 - 1./dy2 );
    Acoef[p][3] += half * 1./dx2;
    Acoef[p][5] += half * 1./dy2;
    b[p] += quarter * c0*exp( tau * phi[p] );

    p = pid(i,j+1);
    Acoef[p][1] += half * ( -1./dx2 - 1./dy2 );
    Acoef[p][2] += half * 1./dx2;
    Acoef[p][5] += half * 1./dy2;
    b[p] += quarter * c0*exp( tau * phi[p] );

}

    ANNOTATE_TASK_END();

    // Apply BCs for SA
    ANNOTATE_TASK_BEGIN(SABC);
    #pragma omp barrier

    for (int pt = Lower; pt <= Upper; ++pt)

```

```

{
  p = pt;
  j = (p-1) / nRealy ;
  i = (p-1) - j*nRealx; j+=1; i+=1;

  if ( myMPI.iPE == 0 ) if ( i == 1 )
    ApplyBC( Acoef , b, p, BCfunc(i,j), Lower, Upper);
  else if ( myMPI.iPE == myMPI.nPEx-1 ) if ( i == nRealx )
    ApplyBC(Acoef , b, p, BCfunc(i,j), Lower, Upper );
  if ( myMPI.jPE == 0 ) if ( j == 1 )
    ApplyBC(Acoef , b, p, BCfunc(i,j), Lower, Upper );
  else if ( myMPI.jPE == myMPI.nPEy-1 ) if ( j == nRealy )
    ApplyBC( Acoef , b, p, BCfunc(i,j), Lower, Upper );
}

  ANNOTATE_TASK_END();

  // Compute right-hand side, f, for Newton system J*delta = -f. Note that f = A*phi - b
  if (solver == "nr"){

    // Loop over each cell, and let each cell contribute to its points' Jacobian
    ANNOTATE_TASK_BEGIN(Jacobian);
    #pragma omp barrier

    for (int pt = Lower ; pt <= Upper ; ++pt )
    {
      p = pt;
      j = (p-1) / nRealy ;
      i = (p-1) - j*nRealx; j+=1; i+=1;

      if(i==nRealx or j==nRealy) continue;

      Jacobian[p][1] += half * ( -1./dx2 - 1./dy2 ) - quarter * tau * c0 * exp( tau * phi[p] );
      Jacobian[p][2] += half * 1./dx2;
      Jacobian[p][4] += half * 1./dy2;

      p = pid(i+1, j);
      Jacobian[p][1] += half * ( -1./dx2 - 1./dy2 ) - quarter * tau * c0 * exp( tau * phi[p] );
      Jacobian[p][3] += half * 1./dx2;
      Jacobian[p][4] += half * 1./dy2;

      p = pid(i+1, j+1);
      Jacobian[p][1] += half * ( -1./dx2 - 1./dy2 ) - quarter * tau * c0 * exp( tau * phi[p] );
      Jacobian[p][3] += half * 1./dx2;

```

```

Jacobian[p][5] += half * 1./dy2;

p = pid(i, j+1);
Jacobian[p][1] += half * ( -1./dx2 - 1./dy2 ) - quarter * tau* c0 * exp( tau * phi[p] );
Jacobian[p][2] += half * 1./dx2;
Jacobian[p][5] += half * 1./dy2;

}

ANNOTATE_TASK_END();
ANNOTATE_TASK_BEGIN(minusf);

vecLOOP
{
minusf[row] = b[row];
colLOOP minusf[row] -= Acoef[row][col] * phi[ Jcoef[row][col] ] ;
}
ANNOTATE_TASK_END();

// Apply BCs for NR
ANNOTATE_TASK_BEGIN(NRBC);
#pragma omp barrier

for (int pt = Lower ; pt <= Upper ; ++pt )
{
p = pt;
j = (p-1) / nRealy ;
i = (p-1) - j*nRealx; j+=1; i+=1;

if ( myMPI.iPE == 0 ) if ( i == 1 )

{ApplyBC( Jacobian , minusf, p, phi[p] - BCfunc(i,j), Lower, Upper ); }
else if ( myMPI.iPE == myMPI.nPEx-1 ) if ( i == nRealx )

{ApplyBC( Jacobian , minusf, p, phi[p] - BCfunc(i,j), Lower, Upper ); }
if ( myMPI.jPE == 0 ) if ( j == 1 )

{ApplyBC( Jacobian , minusf, p, phi[p] - BCfunc(i,j), Lower, Upper ); }
else if ( myMPI.jPE == myMPI.nPEy-1 ) if ( j == nRealy ) {

ApplyBC( Jacobian , minusf, p, phi[p] - BCfunc(i,j), Lower, Upper ); }

}

ANNOTATE_TASK_END();

```

```

}
// debugging

//    if ( printWhat == "printLinear") printLinearSystem();
//    if ( printWhat == "printNewton") printNewtonSystem();
ANNOTATE_SITE_END();

}

void ApplyBC(VDD &Matrix , VD &RHS , int BCrow,double BCvalue, int Lower, int Upper)
{

cLOOP Matrix[BCrow][c] = 0.;
cLOOP Jcoef [BCrow][c] = 0 ;

Matrix[ BCrow ] [ 1 ] = 1.      ;
Jcoef [ BCrow ] [ 1 ] = BCrow   ;
RHS   [ BCrow ]          = BCvalue ;

ANNOTATE_SITE_BEGIN(setBC);
ANNOTATE_ITERATION_TASK(BCrowLoop);
vecLOOP
    if ( row != BCrow )
{
cLOOP
{
    if ( Jcoef[row][c] == BCrow )
{
RHS[row] -= Matrix[row][c]*BCvalue;
Jcoef[row][c] = 0; Matrix[row][c] = 0.;
}
}
}
}
}
ANNOTATE_SITE_END();
}

```

CG and Its Member Functions (The only major difference in the member functions is sizeLOOP instead of rowLOOP)

```

// ==// ||
// || Utility routine: Dot
// ||
// ||
// || Returns the dot product of vec1 and vec2, where vec1 and vec2 are complete

```

```

//  || on each PE.
//  ||
//  ==

double Dot(VD &vec1, VD &vec2 , mpiInfo &myMPI, int size)
{
    ANNOTATE_SITE_BEGIN(Dot);
    double sum = 0.;

    // peMulticity = 1 except on PE boundaries where it is equal to the
    // number of PEs containing the given node.
    ANNOTATE_TASK_BEGIN(Sum);
    sizeLOOP sum += vec1[row]*vec2[row] / myMPI.peMultiplicity[row];
    ANNOTATE_TASK_END();
    // Sum results across PEs and share that sum with all PEs

    double globalSum;
    MPI_Allreduce(&sum, &globalSum, 1 , MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    ANNOTATE_SITE_END();
    return globalSum;
}

//  ==
//  ||
//  || Utility routine: MatVecProd
//  ||
//  || Computes the matrix-vector product, prod = A*p where p is complete
//  || on each PE but A must be summed on PE boundaries.
//  ||
//  ==

void MatVecProd(VDD &Matrix , VD &p , VD &prod , mpiInfo &myMPI, int size, int Lower)
{
    // Serial computation on this PE
    ANNOTATE_SITE_BEGIN(MatVecProd);
    ANNOTATE_TASK_BEGIN(MainLoop);
    sizeLOOP
    {
        prod[row] = 0.;
        colLOOP
    {

        int Acol = Jcoef[row+Lower-1][col];
    }
    }
}

```

```

    if ( Acol > 0 ) prod[row] += Matrix[row][col] * p[ Acol ];
}
    }
    ANNOTATE_TASK_END();
    // Handle PE boundaries

    myMPI.PEsum(prod);
    ANNOTATE_SITE_END();
}

// ==
// ||
// || Utility routine: Residual
// ||
// || Computes the residual, residual = b - A*Sol, where Sol is complete on
// || each PE but b and A must be summed at PE boundaries.
// ||
// ==

void Residual(VDD &Matrix , VD &residual , VD &Sol , VD &RHS, mpiInfo &myMPI, int size, int Lower)
{
    ANNOTATE_SITE_BEGIN(Residual);
    MatVecProd(Matrix, Sol , residual , myMPI, size, Lower);

    ANNOTATE_TASK_BEGIN(rowLOOP);
    sizeLOOP residual[row] = RHS[row] - residual[row];
    ANNOTATE_TASK_END();
    ANNOTATE_SITE_END();
}

// ==
// ||
// || CG (Conjugate Gradient)
// ||
// || Solves the system using Conjugate-Gradient iteration.
// ||
// || See https://en.wikipedia.org/wiki/Conjugate\_gradient\_method
// ||
// ==

void CG(VDD &Matrix , VD &RHS , VD &Solution , mpiInfo & myMPI, int Lower, int Upper, int size)
{
    VD rnew; rnew.resize(nField + 1);

```

```

VD r;      r.resize(nField + 1);
VD p;      p.resize(nField + 1);
VD Ap;     Ap.resize(nField + 1);

double p_dot_Ap;      // Stores matrix-vector product
double r_dot_r;       // Stores r dot r
double rnew_dot_rnew; // Stores rnew dot rnew
double alpha;         // Alpha in the above-referenced algorithm
double beta;          // Beta " " " " " "
double cur_delta;
double tol            = 1.e-15;
int global_converged = 0;
int iter             = 0;
int max_iter         = nField * 0.5;
int converged;

// (1) Initial guess and other initializations

sizeLOOP p[row] = r[row] = Ap[row] = rnew[row] = 0.;

Solution [0] = 0.;
p [0] = 0.;
r [0] = 0.;

// (2) Prepare for parallel computations on RHS

VD b_PESum ;
b_PESum.resize(nField + 1 ) ;
sizeLOOP b_PESum[row] = RHS[row];
myMPI.PESum(b_PESum);

// (3) Initialize residual, r, and r dot r for CG algorithm

Residual(Matrix , r , Solution,b_PESum,myMPI, size, Lower);
sizeLOOP p[row] = r[row];

r_dot_r = Dot(r,r,myMPI, size);

// (4) CG Iterations

ANNOTATE_SITE_BEGIN(CG_iterations);

while ( global_converged == 0 && ++iter <= max_iter)
{

```



```

// (4.1) Compute alpha

ANNOTATE_TASK_BEGIN(MatVecProd);
MatVecProd(Matrix,p,Ap,myMPI, size, Lower);      // A*p (stored in Ap)
ANNOTATE_TASK_END();

ANNOTATE_TASK_BEGIN(Dot);
p_dot_Ap = Dot(p,Ap,myMPI, size); // p*Ap
ANNOTATE_TASK_END();

alpha      = r_dot_r / p_dot_Ap;

// (4.2) Update solution and residual
ANNOTATE_TASK_BEGIN(R_Update);
sizeLOOP {Solution [row] = Solution[row] + alpha *  p[row]; rnew      [row] = r[row]

// (4.3) Compute beta

rnew_dot_rnew = Dot(rnew,rnew,myMPI, size);
beta          = rnew_dot_rnew / r_dot_r;

// (4.4) Update search direction
sizeLOOP {p[row] = rnew[row] + beta*p[row]; r[row] = rnew[row];}

// (4.5) r "new" will be r "old" for next iteration

r_dot_r      = rnew_dot_rnew;
ANNOTATE_TASK_END();

// (4.6) Check convergence on this PE

ANNOTATE_TASK_BEGIN(Check_Converge);

if ( fabs(r_dot_r) < tol)
converged = 1;
else
converged = 0;
ANNOTATE_TASK_END();

// (4.7) Check convergence across PEs, store result in "global_converged"

MPI_Allreduce(&converged, &global_converged, 1 , MPI_INT, MPI_MIN, MPI_COMM_WORLD);

```

```

    }
  //}
  ANNOTATE_SITE_END();

  // (5) Done - Inform user

  if ( global_converged == 1 ) if ( myMPI.myPE == 0 ) cout << "  (o) CG converged in " << iter << endl;
  if ( global_converged == 0 ) if ( myMPI.myPE == 0 ) cout << "  (o) CG failed to converge after " << iter << endl;
}

```