HPSC Lab 7

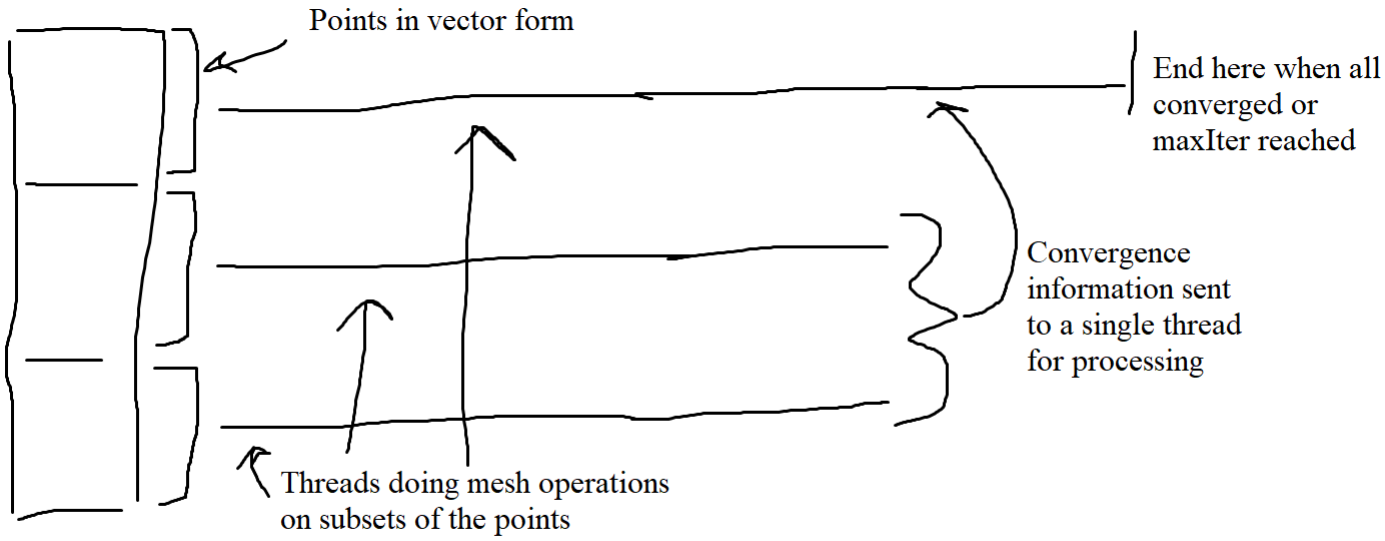
Luna McBride, Jack McDonald

October 20 2023

1 OpenMP Strategy Discussion

It is now time to look into different methods of working with parallelization with the grid itself with this lab. In previous labs, the grid was mostly built for us via the LaplacianOnGrid function, where it was just eating up a lot of time in itself. This lab thus has more of a focus on trying to speedup methods originally given and planting the seeds that there are better ways to leverage the capabilities afforded by a high-performance scientific computing environment. In this case, it starts with threading.

The method of openMP threading in omp.cpp is quite similar to normal multi-threaded applications, except some things are nicely abstracted by the pragma directives. The way that the parallelism is structured in this program is dividing up global matrices and vectors into thread specific matrices and vectors that divide up the problem space. This is an effective way of solving the problem, and to know the overall problem converged, we check if each of the threads have converged since we have broken down our problem space to thread-specific problems. The parallel region begins after all global variables have been defined that we will partition by thread. The only shared variable between threads (besides global variables) is the numTHcoverged variable. We must compute the bounds for each thread in relation too the global variables, then declare local ones that will eventually be plugged back into the global system. We then begin to solve for the local thread-specific system and perform Jacobi iterations until we find whether or not a thread converged. However, before our Jacobi iterations we use two important openMP directives: pragma omp barrier, and pragma omp single. Barrier ensures that we wait for all threads in the program to reach that point before we can continue, and single ensures that only one thread executes the code within its brackets. In this case, the pragma omp single directive makes one thread initialize numTHconverged, because this is the one variable which is shared amongst threads. A rough image showing the thread division, pragma omp single having convergence tests done on a single thread, and end conditions is shown below.

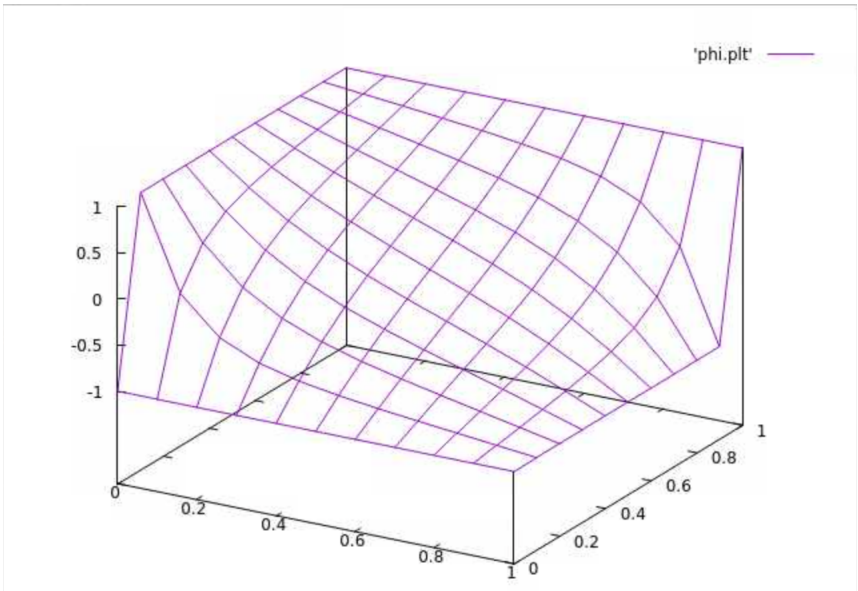


2 Speed-Up Study

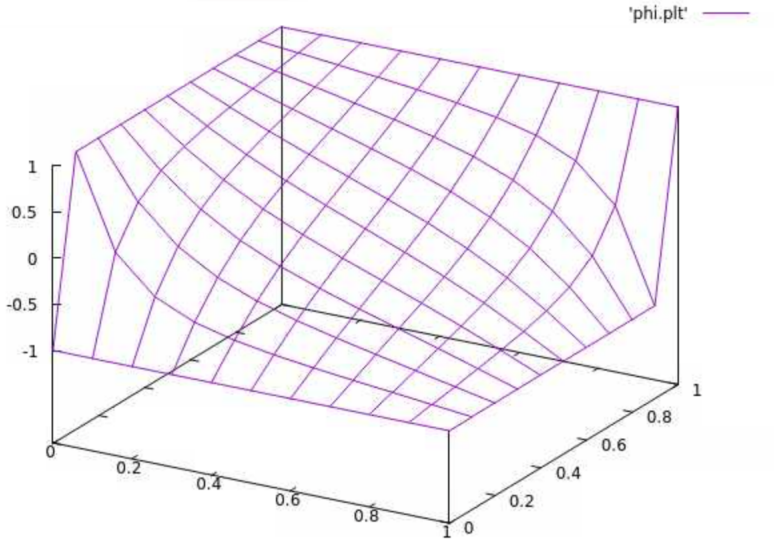
Our speedup study yielded some very interesting results, and ultimately led to our identification of a bottleneck in the program. What we found is that increasing the number of threads increased execution time, rather than decreasing. Using a larger mesh size, it made the increase in time even more significant. What we hypothesized is that, since the parallel section is not optimized and we are doing work on global variables and other things that should not be done in the parallel section, this actually slows down the multi-threaded application versus the single-threaded one. There were also two other key points in the code that significantly increased run times, but these were simple flaws in the code rather than parallelization errors.

This significant slow down on two mesh sizes can be seen below:

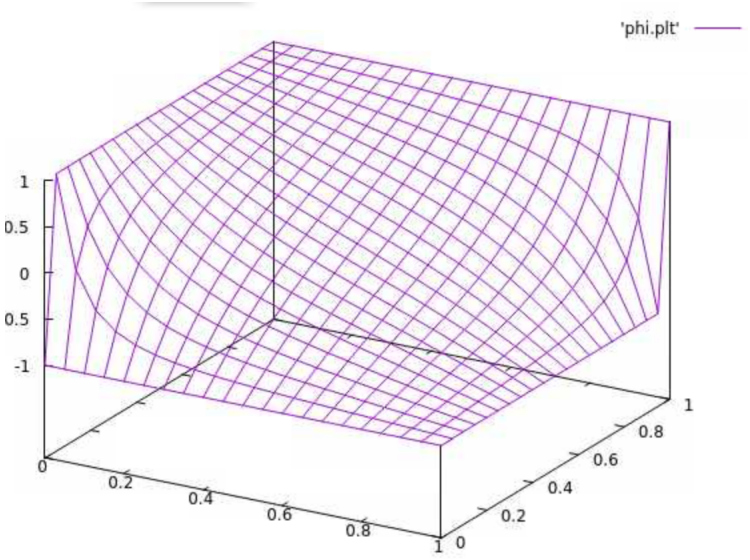
```
[jamc5396@c3cpu-a2-u34-1 omp_02]$ ./omp -nCell 10 -nTH 1
Jacobi converged in 166 iterations.[ main ] EXECUTION TIME  = 1143 (ms)
```



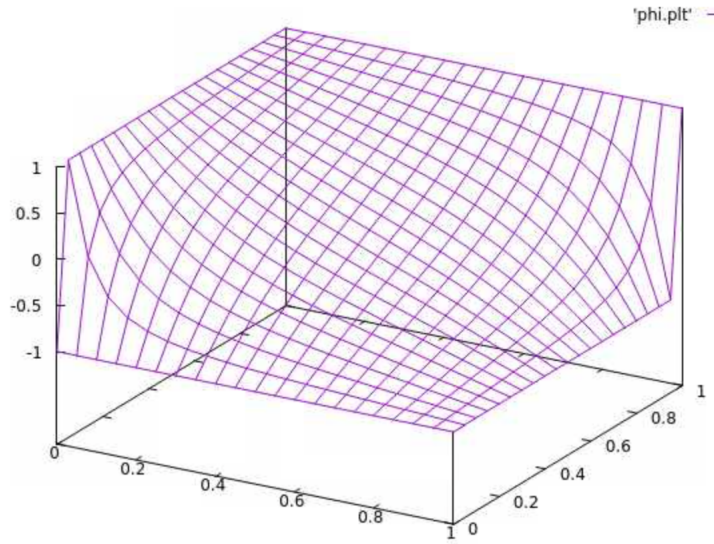

```
[jamc5396@c3cpu-a2-u34-1 omp_02]$ ./omp -nCell 10 -nTH 4
Jacobi converged in 357 iterations.[ main ] EXECUTION TIME  = 16677 (ms)
```




```
[jamc5396@c3cpu-a2-u34-1 omp_02]$ ./omp -nCell 20 -nTH 1
Jacobi converged in 630 iterations.[ main ] EXECUTION TIME  = 9534 (ms)
```




```
[jamc5396@c3cpu-a2-u34-1 omp_02]$ ./omp -nCell 20 -nTH 4
Jacobi converged in 1108 iterations.[ main ] EXECUTION TIME  = 37210 (ms)
```



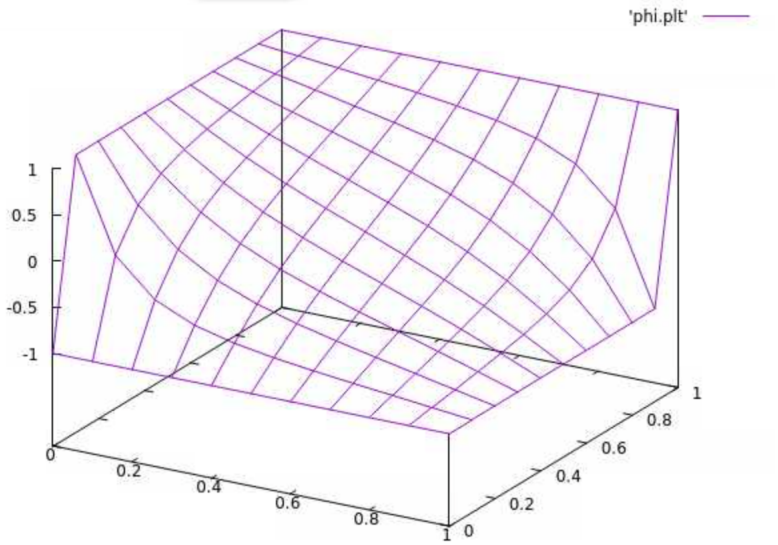
3 Attempted Improvement and Results

3.1 Parallel For Loop Fix

To improve the code, we identified a for-loop initializing our global phi array within the parallel section. Since for-loops come with some overhead, we were essentially multiplying the overhead by the number of threads, rather than only having this overhead once in the non-parallel section. To address this, we simply moved our initialization of phi to outside of the parallel region, shortly after its declaration.

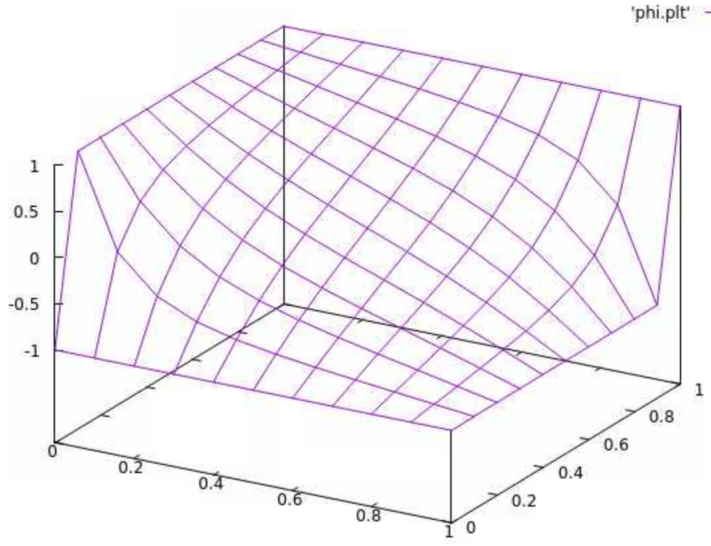
See below, the runtime and graph for omp.cpp with nCell: 10 and nTH: 4 without the code improvement:

```
[jamc5396@c3cpu-a2-u34-1 omp_02]$ ./omp -nCell 10 -nTH 4
Jacobi converged in 357 iterations.[ main ] EXECUTION TIME  = 16677 (ms)
```



See below, the runtime and graph for the same parameters with our code improvement:

```
[jamc5396@c3cpu-a2-u34-1 omp_02]$ ./omp -nCell 10 -nTH 4
Jacobi converged in 345 iterations.[ main ] EXECUTION TIME  = 7695 (ms)
```



Note that there has been an obvious speedup of over 2x, as well as no sacrifice of correctness, as we can see our plots are still completely identical.

3.2 Other Notable Speedup

There were also a few sections of the code that simply dragged everything down for the sake of dragging everything down. To start, there a few lines of if statements that all set the grid to the same values, shown below:

```
if ( i == 0 )  Acoef[pt_TH][0] = 1. ; Jcoef[pt_TH][0] = pt; b[pt_TH] = 1.;
if ( i == nPtsx-1 )  Acoef[pt_TH][0] = 1. ; Jcoef[pt_TH][0] = pt; b[pt_TH] = -1.;
if ( j == 0 )  Acoef[pt_TH][0] = 1. ; Jcoef[pt_TH][0] = pt; b[pt_TH] = -1.;
if ( j == nPtsy-1 )  Acoef[pt_TH][0] = 1. ; Jcoef[pt_TH][0] = pt; b[pt_TH] = 1.;
```

These set the i and j values at the edges specifically, but it all comes to the same value. Simply changing if to else if on the end 3 ifs provides the same effect without having to have the loops check every single case every single time. The resulting code looks as follows:

```
if ( i == 0 )  Acoef[pt_TH][0] = 1. ; Jcoef[pt_TH][0] = pt; b[pt_TH] = 1.;
else if ( i == nPtsx-1 )  Acoef[pt_TH][0] = 1. ; Jcoef[pt_TH][0] = pt; b[pt_TH] = -1.;
else if ( j == 0 )  Acoef[pt_TH][0] = 1. ; Jcoef[pt_TH][0] = pt; b[pt_TH] = -1.;
else if ( j == nPtsy-1 )  Acoef[pt_TH][0] = 1. ; Jcoef[pt_TH][0] = pt; b[pt_TH] = 1.;
```

There is another small time save in this came function, which stems from this original code:

```
Acoef[pt_TH][0] = -4./dh/dh; Jcoef[pt_TH][0] = pt;
Acoef[pt_TH][1] = 1./dh/dh; Jcoef[pt_TH][1] = pt - 1;
Acoef[pt_TH][2] = 1./dh/dh; Jcoef[pt_TH][2] = pt + 1;
Acoef[pt_TH][3] = 1./dh/dh; Jcoef[pt_TH][3] = pt + nPtsx;
Acoef[pt_TH][4] = 1./dh/dh; Jcoef[pt_TH][4] = pt - nPtsx;
```

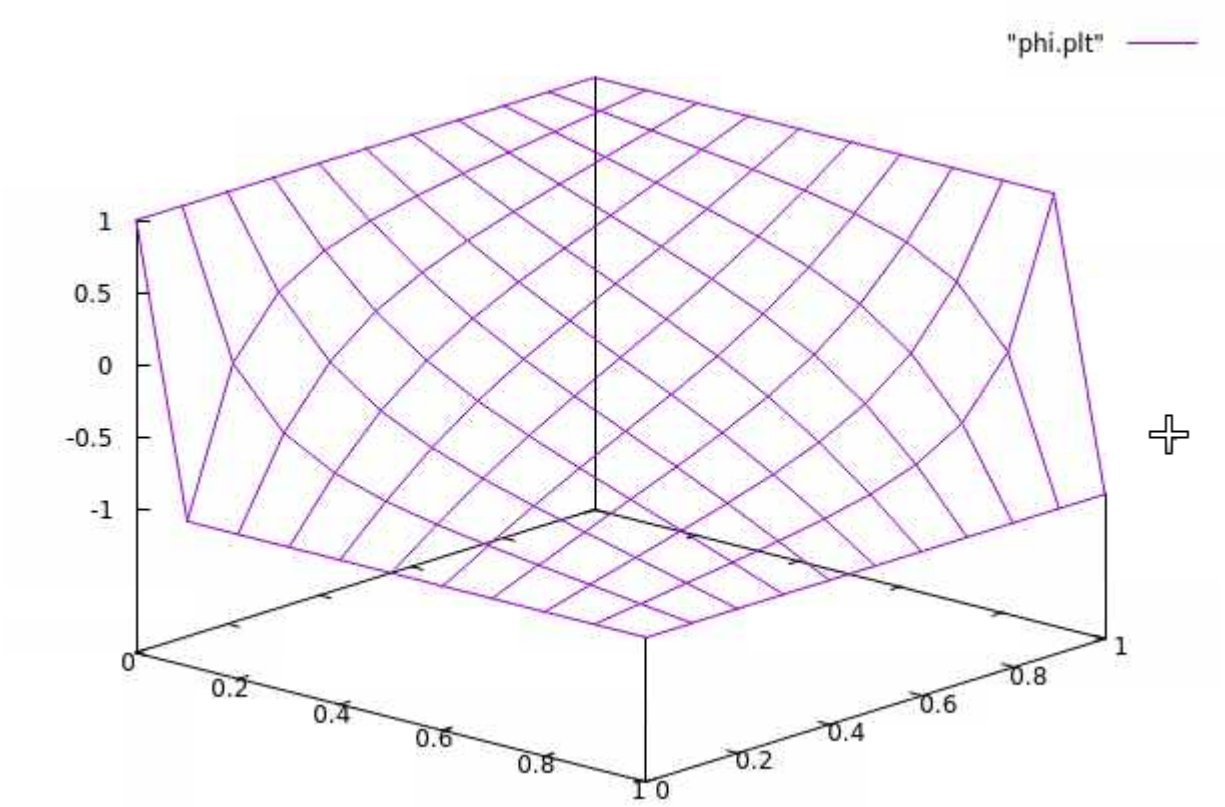
This code is doing the calculation 1./dh/dh four times, which becomes 4*n for the number of times the loop is done. This can be changed as such:

```
Acoef[ pt_TH ][0] = -4./dh/dh; Jcoef[pt_TH ][0] = pt;
Acoef[ pt_TH ][1] = Acoef[ pt_TH ][2] = Acoef[ pt_TH ][3] = Acoef[ pt_TH ][4] = 1./dh/dh;
Jcoef[ pt_TH ][1] = pt - 1;
Jcoef[ pt_TH ][2] = pt + 1;
Jcoef[ pt_TH ][3] = pt + nPtsx;
Jcoef[ pt_TH ][4] = pt - nPtsx;
```

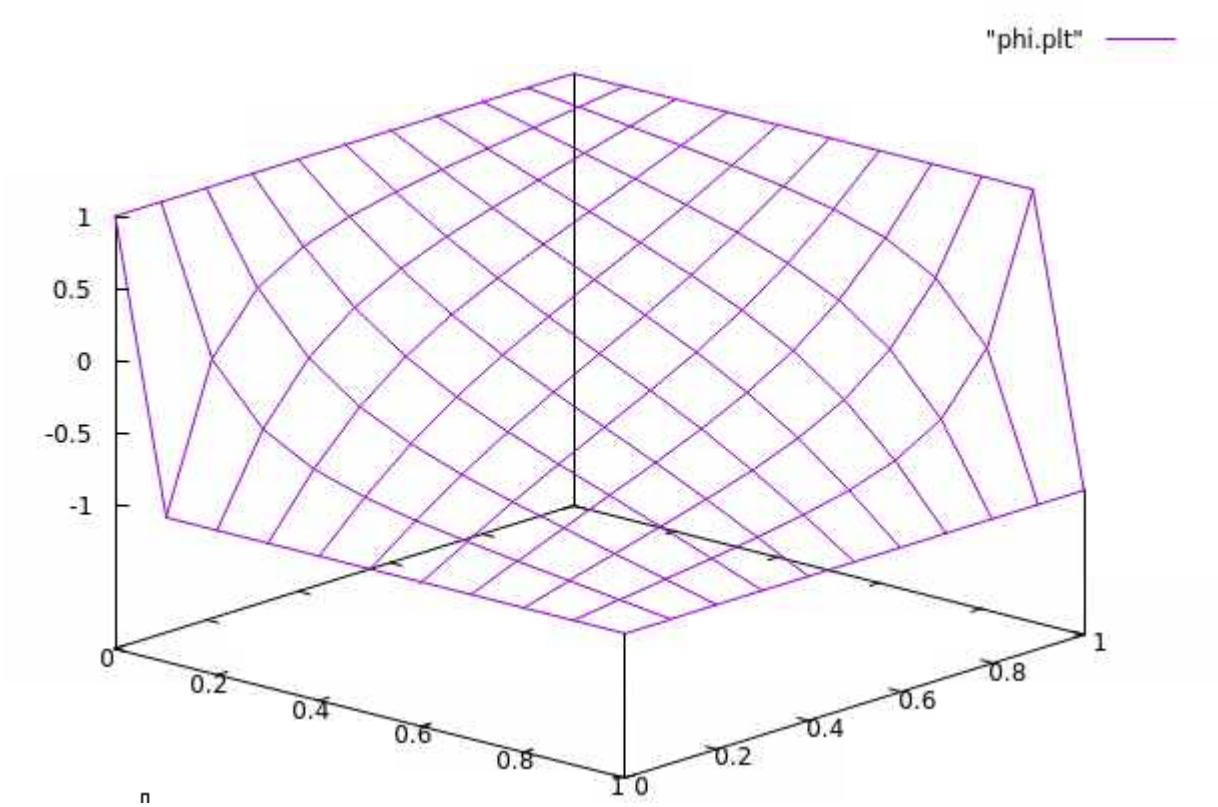
This may not be as pretty, but it reduces unnecessary calculations.

There was one more change that could be made that removed a for loop at the end, and that was moving phi[row] = phiNew[row_TH]; from its own loop to the original main loop of the Jacobi, cutting down the number of loops. All of these loops together resulted in the following speedup:

```
[trmc7708@c3cpu-c9-u5-2 omp_02]$ ./omp -nCell 10 -nTH 1
Jacobi converged in 87 iterations.[ main ] EXECUTION TIME  = 899 (ms)
```




```
[trmc7708@c3cpu-c9-u5-2 omp_02]$ ./omp -nCell 10 -nTH 4
Jacobi converged in 175 iterations.[ main ] EXECUTION TIME  = 14613 (ms)
```



These changes show a speedup of 244 ms for 10 cells 1 thread and 2064 ms for 10 cells 4 threads on its own, not including the fix from above.

4 Self Reflection

4.1 Luna

Interestingly, I could not get the phi for loop time consistently on my computer, instead getting lower times like those shown very inconsistently. This was occurring even without changing Jack's code at all. I even tried putting a pragma single around it to ensure a single process was running it, but I could not reliably reproduce that time save on my system, hence why my part of the speedup emphasized the times compared to the baseline. I think this may be system specific despite me doing it on the on-demand console, as the time save in the timing methods that the professor send out also did nothing for me despite him emphasizing it as a method to save time. I did end up getting a time around what Jack saw, but it occurred only after around 25 runs. I did not add it to my part of the speedup section because of that unreliability, along with the fact that I got a run that took double the normal time within that same set of runs. This shows a slight chance of both really good and very bad runs on my system, thus making them outliers.
As for the actual coding part, I had done the majority of the main code before class on Tuesday and forgot to email Jack that I had, thus leading to him doing it too. We used his as the base for the speedup, but my lack of communication there lead to extra unnecessary work. I note this for my own sake for the future. It was also interesting to see what methods I could use to cause speedup

after reflecting on Lab 6, as some of my logic was incorrect in hindsight. I had done additional attempts of speedup here that all failed following my original logic of pulling items into a new variable to prevent repetition, but that just slowed things down. The aCoef[pt_TH][1] = aCoef... etc = 1./dh/dh speedup is a remnant of this, as doing this provided more speedup than trying to get the 1./dh/dh into a variable. This is unlike how I previously thought it worked and showed a lot better here.

4.2 Jack

For me, what went well in the lab was writing code. I found this to be the easiest and most engaging part of the lab. What was difficult was running tests and making actual hypotheses about the program, its structure, and its logical flow. Similarly, I found speeding up the program easy since I approached it from a coding standpoint. However, when actually interpreting the speedup and realizing that I don't actually understand exactly how the program works, I found a lot more difficulty. Next time, I will spend a lot more time analyzing the program before immediately attempting to jump into filling out todo's. I think this would not only improve my performance on labs, but help me garner a greater understanding of what's going on as well.

5 Appendix A: Original Source Code

```
#include "omp.h"

// ==
// ||
// ||  OMP Laplacian Demo Code
// ||
// ==


int main(  int argc, char *argv[] )
{

  // Start the timer for the entire execution

  struct timespec t0, t1;
  StartTimer(t0);

  // Default values for user input

  int nCell = 190;  // nCell is for the x- and y-directions
  int numTH = 1;    // Number of threads

  // Gather user input from the command line
  for (int count =  0 ; count < argc; ++count)
```

```
    {
      if ( !strcmp(argv[count],"-nCell"    ) ) nCell     = atoi(argv[count+1]);
      if ( !strcmp(argv[count],"-nTH"      ) ) numTH     = atoi(argv[count+1]);
    }

  // Set the number of threads for this execution

  omp_set_num_threads(numTH);

  // Set up grid and linear system information

  double dh          = 1./nCell;        // Size of each cell in each direction (dx and dy)
  int nPtsx          = nCell+1;         // Number of points in the x-direction
  int nPtsy          = nCell+1;         //   "    "    "    "   "  y-    "
  int nField         = nPtsx * nPtsy;   // Total number of points (and number of rows in the linea
  int bandwidth      = 5;               // Number of columns in the sparse-matrix format
  int max_iter       = nField*10;       // Number of allowed Jacobi iterations
  int numTHconverged = 0;               // Number of threads converged

  double   * phi;                       // The unknown for which we are solving A*phi = b

  phi     = Array1D_double(nField);     // Acquire memory for this shared variable.

  // This shared array will store either a 0 or 1 for each thread, 0 meaning not congverged
  // 1 meaning converged.

  int *THconverged = new int [numTH];

// ============================================================================== //
// ======================         BEGIN PARALLEL REGION          ================== //
// ============================================================================== //

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

    // ---------------------------------------------------------------------------
    // (1) Compute this thread's bounds in the global system:
    //
    //     numPerTH = number of rows per thread (the last thread may be slightly different)
    //     Lower    = the first row in the global matrix to be handled by this thread
    //     Upper    = the last   "  "   "    "      "    "  "    "     "   "      "
    // ---------------------------------------------------------------------------

    int numPerTH = nField / numTH;
```

```
  int Lower    = numPerTH * myTH;
  int Upper    = Lower + numPerTH - 1;

  //  (1.1) Adjust Upper for the last (highest numbered) thread to ensure all the rows of the gl

  if ( myTH == numTH-1) Upper = nField - 1;

  // ------------------------------------------------------------------------------------
  // (2) Aquire memory for this thread
  //
  //     nField_TH = the number of field variables (phi) to be handled by this thread
  // ------------------------------------------------------------------------------------

  int nField_TH = Upper - Lower + 1;

  double * * Acoef;
  int    * * Jcoef;
  double   * b;
  double   * phiNew;
  //     Acoef, Jcoef, b, and phiNew are not the global matrices and arrays; here they contain o
  //     rows being handled by this thread.

  Acoef    = Array2D_double(  nField_TH, bandwidth  );
  Jcoef    = Array2D_int   ( nField_TH, bandwidth   );
  b        = Array1D_double( nField_TH  );
  phiNew   = Array1D_double( nField_TH   );

  // ------------------------------------------------------------------------------------
  // (3) Initialize the linear system and form the initial guess for phi
  // ------------------------------------------------------------------------------------

  for ( int row = 0 ; row < nField_TH ; ++row )
    {
      for ( int col = 0 ; col < bandwidth ; ++col )
        {
          Acoef[row][col] = 0.;
          Jcoef[row][col] = 0 ;
}
      b[row] = 0.;
    }

  for ( int row = Lower ; row <= Upper ; ++row )
    {
      phi[row] = 0.;
    }
```

```
    // --------------------------------------------------------------------------------
    // (4) Form the linear system.  Here "pt" represents "point" number in the mesh.  It is
    //     equal to the row number in the linear system, too.
    // --------------------------------------------------------------------------------

    for ( int pt = Lower ; pt <= Upper ; ++pt )
      {
// Using the same logic as for converting myPE to iPE and jPE,
        // compute the i,j logical coordinates of "pt" in the mesh:

        //int j = pt % nCell;
        int j = pt / nPtsy;
        int i = pt - j*nPtsx;
  //int i = pt / nPtsy;

        // Compute the row number local to this thread, relative to its Acoef/Jcoef arrays

        int pt_TH = pt - Lower;

        // Populate the linear system for all interior points using that local row number

        if ( i > 0 && i < nPtsx-1 )
  if ( j > 0 && j < nPtsy-1 )
           {
             // The point of these "TO-DOs" is to make sure you understand how to compute the row

             Acoef[ pt_TH ][0] = -4./dh/dh;  Jcoef[pt_TH ][0] = pt;
             Acoef[ pt_TH ][1] =  1./dh/dh;  Jcoef[ pt_TH ][1] = pt - 1;
             Acoef[ pt_TH ][2] =  1./dh/dh;  Jcoef[ pt_TH ][2] = pt + 1;
             Acoef[ pt_TH ][3] =  1./dh/dh;  Jcoef[ pt_TH ][3] = pt + nPtsx;
             Acoef[ pt_TH ][4] =  1./dh/dh;  Jcoef[ pt_TH ][4] = pt - nPtsx;
           }

        // Apply boundary conditions

        // The point of these "TO-DOs" is to make sure you understand how to compute the row numbe
if ( i == 0       ) { Acoef[ pt_TH ][0] = 1. ; Jcoef[ pt_TH ][0] = pt; b[ pt_TH ] =  1.; }
        if ( i == nPtsx-1 ) { Acoef[ pt_TH ][0] = 1. ; Jcoef[ pt_TH ][0] = pt; b[ pt_TH ] = -1.; }
        if ( j == 0       ) { Acoef[ pt_TH ][0] = 1. ; Jcoef[ pt_TH ][0] = pt; b[ pt_TH ] = -1.; }
        if ( j == nPtsy-1 ) { Acoef[ pt_TH ][0] = 1. ; Jcoef[ pt_TH ][0] = pt; b[ pt_TH ] =  1.; }
      }

    // --------------------------------------------------------------------------------
    // (5) Perform Jacobi iterations
```

```
    // ------------------------------------------------------------------------------

    int iter       = 0;
    int thisThreadConverged;

    #pragma omp barrier
    // numTHconverged counts the number of threads converged.  We do not care which thread inital

    #pragma omp single
    {
      numTHconverged = 0;
    }

    // Iterate until all of the threads have converged or we have exceeded the number of allowed i

    while ( numTHconverged < numTH && ++iter <= max_iter )   // The TO-DO is the global convergenc
      {
        thisThreadConverged = 1;

        for ( int row = Lower ; row <= Upper ; ++row )
          {
            int row_TH = row - Lower;                                    // row_TH is the row numbe
    phiNew[ row_TH ] = b[ row_TH ];

            for ( int col = 1 ; col < bandwidth ; ++col ) phiNew[ row_TH ] -= Acoef[ row_TH ][col]

            phiNew[ row_TH ] /= Acoef[ row_TH ][0];

            if ( fabs(phiNew[ row_TH ] -  phi[row] ) > 1.e-10 ) thisThreadConverged = 0;

          }

        // (5.1) Record in shared array if this thread converged or not

        THconverged[myTH] = thisThreadConverged;      // For this TO-DO, enter the variable that s

// (5.2) Count the number of threads that have converged.  We do not care which thread does the co

        #pragma omp barrier

#pragma omp single
        {
          numTHconverged = 0;
          for ( int i = 0 ; i < numTH ; ++i ) numTHconverged += THconverged[i] ;
          if ( numTHconverged == numTH ) printf("Jacobi converged in %d iterations.",iter);
```

```
        }

        // (5.3) Update the shared/global array phi with this thread's values

        for ( int row = Lower ; row <= Upper ; ++row )
          {
            int row_TH = row - Lower;         // row_TH is the row number on this thread.
            phi[row] = phiNew[ row_TH ];
   }
      }

  }

// ================================================================================== //
// ======================           END PARALLEL REGION          ==================== //
// ================================================================================== //

  if ( numTHconverged != numTH ) printf("WARNING: Jacobi did not converge.\n");

  EndTimer("main", t0,t1);

  plot("phi",phi,nPtsx,nPtsy,dh);
  return 0;
}
```

6 Appendix B: Source Code After Improvement

```
#include "omp.h"

// ==
// ||
// ||  OMP Laplacian Demo Code
// ||
// ==


int main(  int argc, char *argv[] )
{

  // Start the timer for the entire execution

  struct timespec t0, t1;
  StartTimer(t0);
```

```
  // Default values for user input

  int nCell = 190;  // nCell is for the x- and y-directions
  int numTH = 1;    // Number of threads

  // Gather user input from the command line
  for (int count =  0 ; count < argc; ++count)
    {
      if ( !strcmp(argv[count],"-nCell"    ) ) nCell     = atoi(argv[count+1]);
      if ( !strcmp(argv[count],"-nTH"      ) ) numTH     = atoi(argv[count+1]);
    }

  // Set the number of threads for this execution

  omp_set_num_threads(numTH);

  // Set up grid and linear system information

  double dh          = 1./nCell;        // Size of each cell in each direction (dx and dy)
  int nPtsx          = nCell+1;         // Number of points in the x-direction
  int nPtsy          = nCell+1;         //   "    "    "    "   "  y-    "
  int nField         = nPtsx * nPtsy;   // Total number of points (and number of rows in the linea
  int bandwidth      = 5;               // Number of columns in the sparse-matrix format
  int max_iter       = nField*10;       // Number of allowed Jacobi iterations
  int numTHconverged = 0;               // Number of threads converged

  double   * phi;                       // The unknown for which we are solving A*phi = b

  phi     = Array1D_double(nField);     // Acquire memory for this shared variable.
  #pragma omp single
  {
    for ( int i = 0 ; i < nField ; i++)
        {
          phi[i] = 0.;
        }
  }

  // This shared array will store either a 0 or 1 for each thread, 0 meaning not congverged
  // 1 meaning converged.

  int *THconverged = new int [numTH];

// ========================================================================== //
// ======================         BEGIN PARALLEL REGION          ================== //
```

```
// ============================================================================== //

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

    // ------------------------------------------------------------------------------
    // (1) Compute this thread's bounds in the global system:
    //
    //     numPerTH = number of rows per thread (the last thread may be slightly different)
    //     Lower    = the first row in the global matrix to be handled by this thread
    //     Upper    = the last   "  "   "    "      "    "  "    "     "   "      "
    // ------------------------------------------------------------------------------

    int numPerTH = nField / numTH;
    int Lower    = numPerTH * myTH;
    int Upper    = Lower + numPerTH - 1;

    //  (1.1) Adjust Upper for the last (highest numbered) thread to ensure all the rows of the gl

    if ( myTH == numTH-1) Upper = nField - 1;

    // ------------------------------------------------------------------------------
    // (2) Aquire memory for this thread
    //
    //     nField_TH = the number of field variables (phi) to be handled by this thread
    // ------------------------------------------------------------------------------

    int nField_TH = Upper - Lower + 1;

    double * * Acoef;
    int    * * Jcoef;
    double   * b;
    double   * phiNew;
    //     Acoef, Jcoef, b, and phiNew are not the global matrices and arrays; here they contain o
    //     rows being handled by this thread.

    Acoef    = Array2D_double(  nField_TH, bandwidth  );
    Jcoef    = Array2D_int   ( nField_TH, bandwidth   );
    b        = Array1D_double( nField_TH  );
    phiNew   = Array1D_double( nField_TH   );

    // ------------------------------------------------------------------------------
    // (3) Initialize the linear system and form the initial guess for phi
    // ------------------------------------------------------------------------------
```

```
    for ( int row = 0 ; row < nField_TH ; ++row )
      {
        for ( int col = 0 ; col < bandwidth ; ++col )
          {
            Acoef[row][col] = 0.;
            Jcoef[row][col] = 0 ;
  }
        b[row] = 0.;
      }

    // ------------------------------------------------------------------------------
    // (4) Form the linear system.  Here "pt" represents "point" number in the mesh.  It is
    //     equal to the row number in the linear system, too.
    // ------------------------------------------------------------------------------

    for ( int pt = Lower ; pt <= Upper ; ++pt )
      {
// Using the same logic as for converting myPE to iPE and jPE,
        // compute the i,j logical coordinates of "pt" in the mesh:

        //int j = pt % nCell;
        int j = pt / nPtsy;
        int i = pt - j*nPtsx;
  //int i = pt / nPtsy;

        // Compute the row number local to this thread, relative to its Acoef/Jcoef arrays

        int pt_TH = pt - Lower;

        // Populate the linear system for all interior points using that local row number

        if ( i > 0 && i < nPtsx-1 )
  if ( j > 0 && j < nPtsy-1 )
            {
              // The point of these "TO-DOs" is to make sure you understand how to compute the row


              Acoef[ pt_TH ][0] = -4./dh/dh;  Jcoef[pt_TH ][0] = pt;
              Acoef[ pt_TH ][1] = Acoef[ pt_TH ][2] = Acoef[ pt_TH ][3] = Acoef[ pt_TH ][4] =  1./
      Jcoef[ pt_TH ][1] = pt - 1;
              Jcoef[ pt_TH ][2] = pt + 1;
              Jcoef[ pt_TH ][3] = pt + nPtsx;
              Jcoef[ pt_TH ][4] = pt - nPtsx;
            }
```

```
        // Apply boundary conditions


        // The point of these "TO-DOs" is to make sure you understand how to compute the row numbe
if (i==0) { Acoef[ pt_TH ][0] = 1. ; Jcoef[ pt_TH ][0] = pt; b[ pt_TH ] =  1.; }
        else if (i == nPtsx-1) { Acoef[ pt_TH ][0] = 1. ; Jcoef[ pt_TH ][0] = pt; b[ pt_TH ] = -1.
        else if ( j==0) { Acoef[ pt_TH ][0] = 1. ; Jcoef[ pt_TH ][0] = pt; b[ pt_TH ] = -1.; }
        else if ( j == nPtsy-1 ) { Acoef[ pt_TH ][0] = 1. ; Jcoef[ pt_TH ][0] = pt; b[ pt_TH ] =
      }

    // ------------------------------------------------------------------------------------
    // (5) Perform Jacobi iterations
    // ------------------------------------------------------------------------------------

    int iter       = 0;
    int thisThreadConverged;

    #pragma omp barrier
    // numTHconverged counts the number of threads converged.  We do not care which thread initali

    #pragma omp single
    {
      numTHconverged = 0;
    }

    // Iterate until all of the threads have converged or we have exceeded the number of allowed i

    while ( numTHconverged < numTH && ++iter <= max_iter )   // The TO-DO is the global convergenc
      {
        thisThreadConverged = 1;

        for ( int row = Lower ; row <= Upper ; ++row )
          {
            int row_TH = row - Lower;                                    // row_TH is the row numbe
    phiNew[ row_TH ] = b[ row_TH ];

            for ( int col = 1 ; col < bandwidth ; ++col ) phiNew[ row_TH ] -= Acoef[ row_TH ][col]

            phiNew[ row_TH ] /= Acoef[ row_TH ][0];

            if ( fabs(phiNew[ row_TH ] -  phi[row] ) > 1.e-10 ) thisThreadConverged = 0;
    phi[row] = phiNew[ row_TH ];
          }
```

```
        // (5.1) Record in shared array if this thread converged or not

        THconverged[myTH] = thisThreadConverged;      // For this TO-DO, enter the variable that s

// (5.2) Count the number of threads that have converged.  We do not care which thread does the co

        #pragma omp barrier

#pragma omp single
        {
          numTHconverged = 0;
          for ( int i = 0 ; i < numTH ; ++i ) numTHconverged += THconverged[i] ;
          if ( numTHconverged == numTH ) printf("Jacobi converged in %d iterations.",iter);
        }

        // (5.3) Update the shared/global array phi with this thread's values

// for ( int row = Lower ; row <= Upper ; ++row )
// {
    // int row_TH = row - Lower;         // row_TH is the row number on this thread.
    // phi[row] = phiNew[ row_TH ];
    // }
}

  }

// ================================================================================== //
// ======================          END PARALLEL REGION          ==================== //
// ================================================================================== //

  if ( numTHconverged != numTH ) printf("WARNING: Jacobi did not converge.\n");

  EndTimer("main", t0,t1);

  plot("phi",phi,nPtsx,nPtsy,dh);
  return 0;
}
```