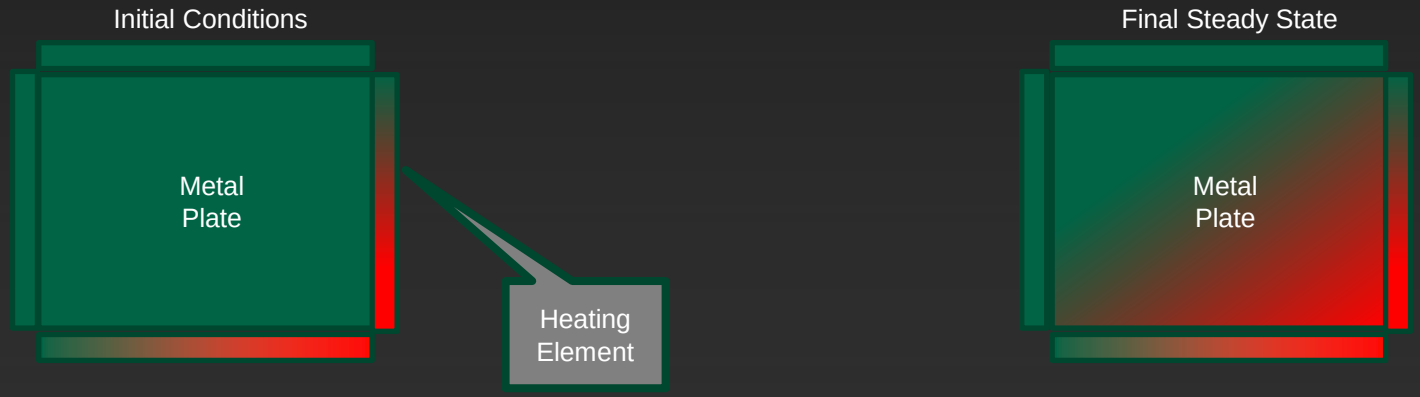
Laplace Exercise

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Our Foundation Exercise: Laplace Solver

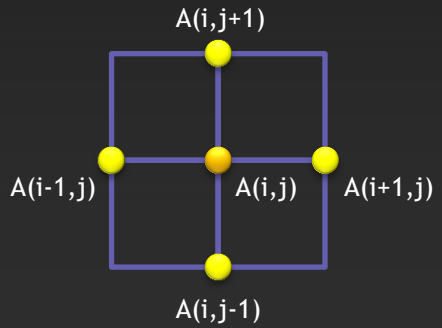
- I've been using this for MPI, OpenMP and now OpenACC. It is a great simulation problem, not rigged for MPI.
- In this most basic form, it solves the Laplace equation: $\nabla^2 f(x, y) = 0$
- The Laplace Equation applies to many physical problems, including:
  - Electrostatics
  - Fluid Flow
  - Temperature

- For temperature, it is the Steady State Heat Equation:



Exercise Foundation: Jacobi Iteration

- The Laplace equation on a grid states that each grid point is the average of its neighbors.
- We can iteratively converge to that state by repeatedly computing new values at each point from the average of neighboring points.
- We just keep doing this until the difference from one pass to the next is small enough for us to tolerate.



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Serial Code Implementation

```
for(i = 1; i <= ROWS; i++) {
     for(j = 1; j <= COLUMNS; j++) {
          Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                      Temperature_last[i][j+1] + Temperature_last[i][j-1]);
     }
}
```

```
do j=1,columns
     do i=1,rows
          temperature(i,j)= 0.25 * (temperature_last(i+1,j)+temperature_last(i-1,j) + &
                                    temperature_last(i,j+1)+temperature_last(i,j-1) )
     enddo
enddo
```

Serial C Code (kernel)

```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

     for(i = 1; i <= ROWS; i++) {
          for(j = 1; j <= COLUMNS; j++) {
               Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                           Temperature_last[i][j+1] + Temperature_last[i][j-1]);
          }
     }

     dt = 0.0;

     for(i = 1; i <= ROWS; i++){
          for(j = 1; j <= COLUMNS; j++){
               dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
               Temperature_last[i][j] = Temperature[i][j];
          }
     }

     if((iteration % 100) == 0) {
          track_progress(iteration);
     }

     iteration++;

}
```

Done?

Calculate

Update temp array and find max change

Output

Serial C Code Subroutines

```
void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[i][j] = 0.0;
        }
    }

    // these boundary conditions never change throughout run

    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
```

BCs could run from 0 to ROWS+1 or from 1 to ROWS. We chose the former.

```
void track_progress(int iteration) {

  int i;

  printf("-- Iteration: %d --\n", iteration);
  for(i = ROWS-5; i <= ROWS; i++) {
     printf("[%d,%d]: %5.2f ", i, i,Temperature[i][i]);
  }
  printf("\n");
}
```

Whole C Code

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

// size of plate
#define COLUMNS    1000
#define ROWS       1000

// largest permitted change in temp (This value takes about 3400 steps)
#define MAX_TEMP_ERROR 0.01

double Temperature[ROWS+2][COLUMNS+2];      // temperature grid
double Temperature_last[ROWS+2][COLUMNS+2]; // temperature grid from last iteration

//   helper routines
void initialize();
void track_progress(int iter);

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

    int i, j;                                            // grid indexes
    int max_iterations;                                  // number of iterations
    int iteration=1;                                     // current iteration
    double dt=100;                                       // largest change in t
    struct timeval start_time, stop_time, elapsed_time;  // timers

    printf("Maximum iterations [100-4000]?\n");
    scanf("%d", &max_iterations);

    gettimeofday(&start_time,NULL); // Unix timer

    initialize();                   // initialize Temp_last including boundary conditions

    // do until error is minimal or until max steps
    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

        // main calculation: average my four neighbors
        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                            Temperature_last[i][j+1] + Temperature_last[i][j-1]);
            }
        }

        dt = 0.0; // reset largest temperature change

        // copy grid to old grid for next iteration and find latest dt
        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
              dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
              Temperature_last[i][j] = Temperature[i][j];
            }
        }

        // periodically print test values
        if((iteration % 100) == 0) {
            track_progress(iteration);
        }

        iteration++;
    }

    gettimeofday(&stop_time,NULL);
    timersub(&stop_time, &start_time, &elapsed_time); // Unix time subtract routine

    printf("\nMax error at iteration %d was %f\n", iteration-1, dt);
    printf("Total time was %f seconds.\n", elapsed_time.tv_sec+elapsed_time.tv_usec/1000000.0);

}

// initialize plate and boundary conditions
// Temp_last is used to to start first iteration
void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[i][j] = 0.0;
        }
    }

    // these boundary conditions never change throughout run

    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}

// print diagonal in bottom right corner where most action is
void track_progress(int iteration) {

    int i;

    printf("---------- Iteration number: %d ------------\n", iteration);
    for(i = ROWS-5; i <= ROWS; i++) {
        printf("[%d,%d]: %5.2f  ", i, i, Temperature[i][i]);
    }
    printf("\n");
}
```

Serial Fortran Code (kernel)

```
do while ( dt > max_temp_error .and. iteration <= max_iterations)

  do j=1,columns
     do i=1,rows
        temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                               temperature_last(i,j+1)+temperature_last(i,j-1) )
     enddo
  enddo

  dt=0.0

  do j=1,columns
     do i=1,rows
        dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
        temperature_last(i,j) = temperature(i,j)
     enddo
  enddo

  if( mod(iteration,100).eq.0 ) then
     call track_progress(temperature, iteration)
  endif

  iteration = iteration+1

enddo
```

Done?

Calculate

Update temp array and find max change

Output

Serial Fortran Code Subroutines

```
subroutine initialize( temperature_last )
      implicit none

      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      integer                        :: i,j

      double precision, dimension(0:rows+1,0:columns+1) :: temperature_last

      temperature_last = 0.0

      !these boundary conditions never change throughout run

      !set left side to 0 and right to linear increase
      do i=0,rows+1
         temperature_last(i,0) = 0.0
         temperature_last(i,columns+1) = (100.0/rows) * i
      enddo

      !set top to 0 and bottom to linear increase
      do j=0,columns+1
         temperature_last(0,j) = 0.0
         temperature_last(rows+1,j) = ((100.0)/columns) * j
      enddo

end subroutine initialize
```

```
subroutine track_progress(temperature, iteration)
      implicit none

      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      integer                        :: i,iteration

      double precision, dimension(0:rows+1,0:columns+1) :: temperature

      print *, '---------- Iteration number: ', iteration, ' ---------------'
      do i=5,0,-1
         write (*,'("("i4,",",i4,"):",f6.2,"  ")',advance='no'), &
                  rows-i,columns-i,temperature(rows-i,columns-i)
      enddo
      print *
```

Whole Fortran Code

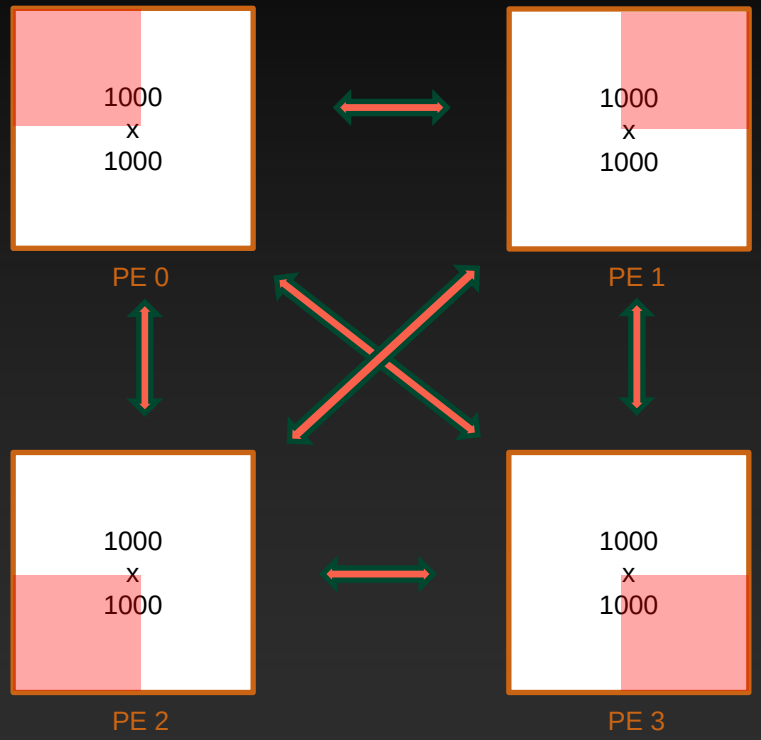
```
program serial
      implicit none

      !Size of plate
      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      double precision, parameter    :: max_temp_error=0.01

      integer                        :: i, j, max_iterations, iteration=1
      double precision               :: dt=100.0
      real                           :: start_time, stop_time

      double precision, dimension(0:rows+1,0:columns+1) :: temperature, temperature_last

      print*, 'Maximum iterations [100-4000]?'
      read*,   max_iterations

      call cpu_time(start_time)      !Fortran timer

      call initialize(temperature_last)

      !do until error is minimal or until maximum steps
      do while ( dt > max_temp_error .and. iteration <= max_iterations)

         do j=1,columns
            do i=1,rows
               temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                                      temperature_last(i,j+1)+temperature_last(i,j-1) )
            enddo
         enddo

         dt=0.0

         !copy grid to old grid for next iteration and find max change
         do j=1,columns
            do i=1,rows
               dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
               temperature_last(i,j) = temperature(i,j)
            enddo
         enddo

         !periodically print test values
         if( mod(iteration,100).eq.0 ) then
            call track_progress(temperature, iteration)
         endif

         iteration = iteration+1

      enddo

      call cpu_time(stop_time)

      print*, 'Max error at iteration ', iteration-1, ' was ',dt
      print*, 'Total time was ',stop_time-start_time, ' seconds.'

end program serial
```

```
! initialize plate and boundery conditions
! temp_last is used to to start first iteration
subroutine initialize( temperature_last )
      implicit none

      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      integer                        :: i,j

      double precision, dimension(0:rows+1,0:columns+1) :: temperature_last

      temperature_last = 0.0

      !these boundary conditions never change throughout run

      !set left side to 0 and right to linear increase
      do i=0,rows+1
         temperature_last(i,0) = 0.0
         temperature_last(i,columns+1) = (100.0/rows) * i
      enddo

      !set top to 0 and bottom to linear increase
      do j=0,columns+1
         temperature_last(0,j) = 0.0
         temperature_last(rows+1,j) = ((100.0)/columns) * j
      enddo

end subroutine initialize

!print diagonal in bottom corner where most action is
subroutine track_progress(temperature, iteration)
      implicit none

      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      integer                        :: i,iteration

      double precision, dimension(0:rows+1,0:columns+1) :: temperature

      print *, '---------- Iteration number: ', iteration, ' ---------------'
      do i=5,0,-1
         write (*,'("("i4,",",i4,"):",f6.2,"  ")',advance='no'), &
                    rows-i,columns-i,temperature(rows-i,columns-i)
      enddo
      print *
end subroutine track_progress
```

First Things First: Domain Decomposition

- All processors have an entire Temperature array.
- Each processor works on its part of Temperature.
- After every iteration, all processors broadcast to the other processors.

- No! Preserving any large shared data structures is bad.
- Increased memory. *Not Scalable.*
- Large messages. *Not Scalable.*

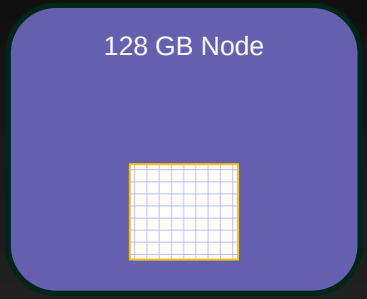


Scalability Demands No Global Arrays

1000 x 1000 Grid

8 bytes * 1M elements * 2 arrays

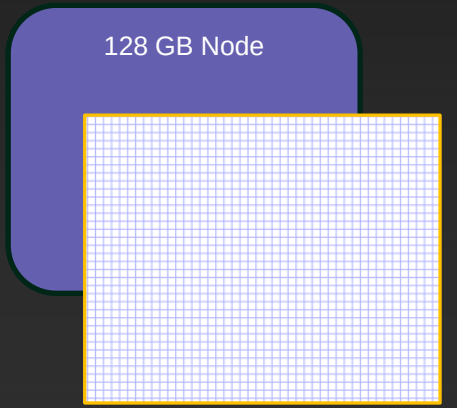
= 16MB memory



100,000 x 100,000 Grid

8 bytes * 10 Billion elements * 2 arrays

= 160 GB memory

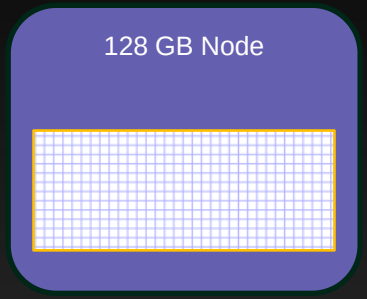


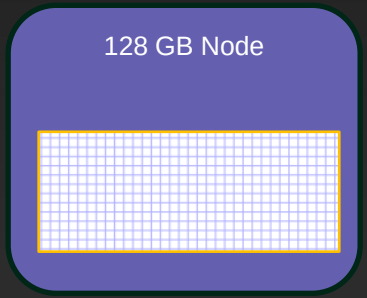
Divide Up Data As Needed

100,000 x 100,000 Grid

8 bytes * 10 Billion elements * 2 arrays

= 160 GB memory





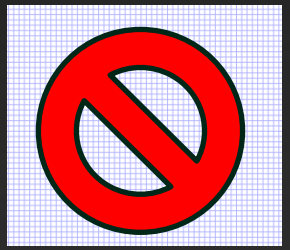
Keep Partitioning Data

1,000,000 x 1,000,000 Grid

8 bytes * 1 Trillion elements * 2 arrays

= 16 TB memory

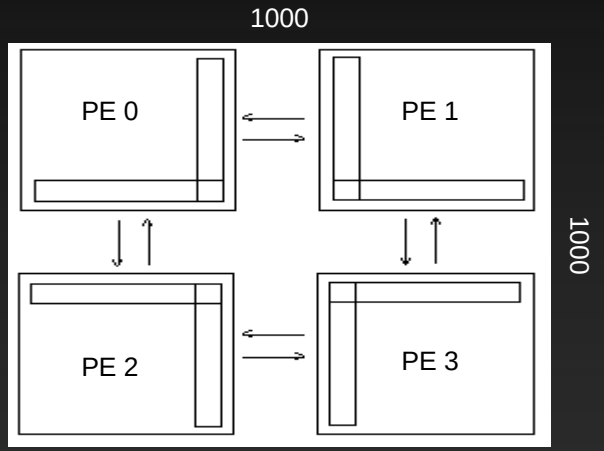
= 126 Regular Bridges Nodes



128 GB Node	128 GB Node	128 GB Node	128 GB Node
128 GB Node	128 GB Node	128 GB Node	128 GB Node
128 GB Node	128 GB Node	128 GB Node	128 GB Node
128 GB Node	128 GB Node	128 GB Node	128 GB Node
128 GB Node	128 GB Node	128 GB Node	128 GB Node
128 GB Node	128 GB Node	128 GB Node	128 GB Node
128 GB Node	128 GB Node	128 GB Node	128 GB Node
128 GB Node	128 GB Node	128 GB Node	128 GB Node
etc.	etc.	etc.	

Try Again: Domain Decomposition II

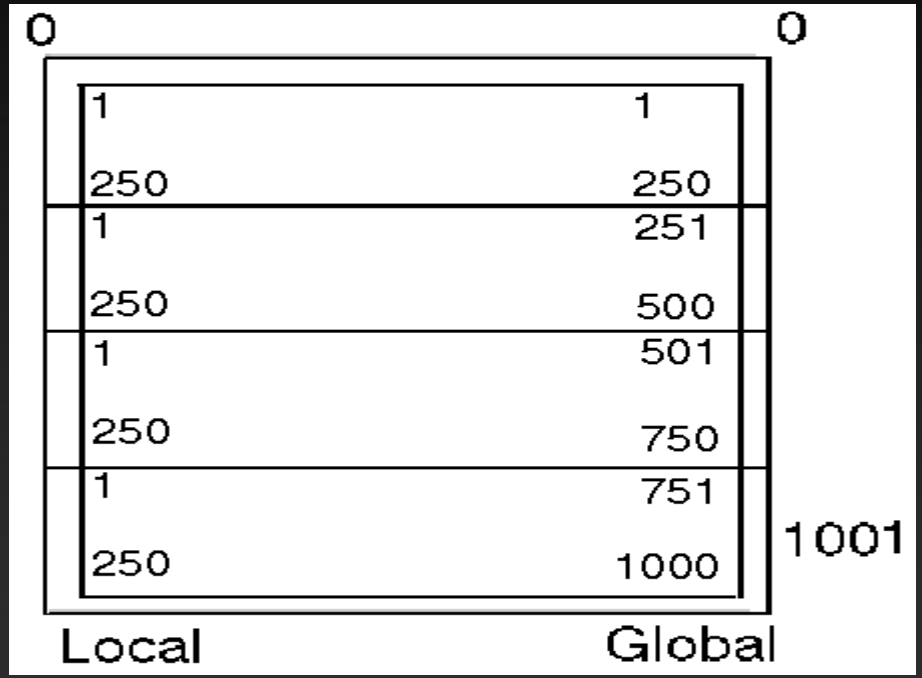
- Each processor has only its own sub-grid.
- 1000x1000 array not longer exists (except on paper).
- Communicate boundary values only.
- Reduces memory.
- Reduces communications.
- Have to keep track of neighbors in two directions.
- But not bad.



Simplest: Domain Decomposition III

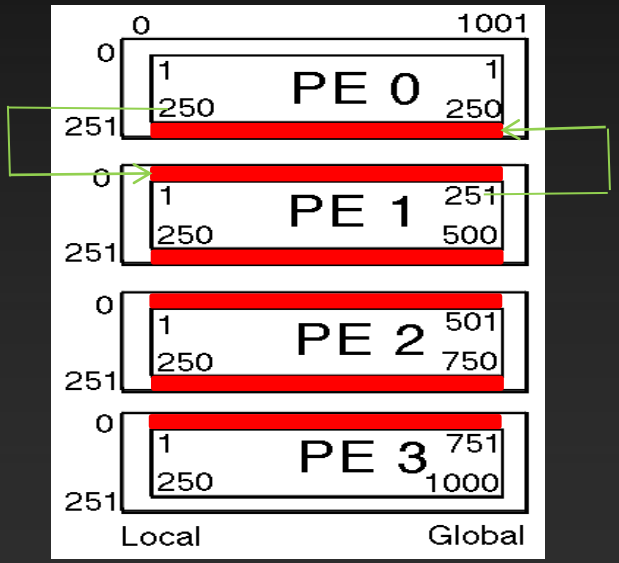
- Only have to keep track of up/down neighbors, and no corner case.
- Easiest with our simple message passing experience.
- Scales, as below. How would we handle 5 PEs with the “square decomposition”?

Simplest Decomposition for C Code



Simplest Decomposition for C Code

In the parallel case, we will break this up into 4 processors. There is only one set of boundary values. But when we distribute the data, each processor needs to have an extra row for data distribution, these are commonly called the "ghost cells".



The program has a local view of data. The programmer has to have a global view of data. The ghost cells don't exist in the global dataset. They are only copies from the "real" data in the adjacent PE.

Sending Multiple Elements

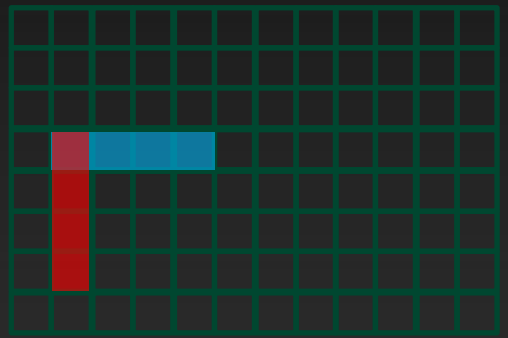
- For the first time we want to send multiple elements. In this case, a whole row or column of data. That is exactly what the count parameter is for.
- The common use of the count parameter is to point the Send or Receive routine at the first element of an array, and then the count will proceed to strip off as many elements as you specify.
- This implies (and demands) that the elements are contiguous in memory. That will be true for one dimension of an array, but the other dimension(s) will have a stride.
- In C this is true for our rows. In Fortran this is true for our columns. This will give us a strong preference for the problem orientation in each language. Then we don't have to worry about strides in the strips that we send.
- However, it is very often necessary to send messages that are not contiguous data. Using defined data types, we can send other array dimensions, or even blocks or surfaces. We will talk about that capability in the Advanced talk.

Sending Multiple Elements

C:

This last index is the one contiguous in memory.

```
int A[8][12];

MPI_Send(&A[3][1], 4, MPI_INT, pe, tag, MPI_COMM_WORLD);
```



Fortran:

```
integer A(0:7,0:11)

MPI_Send(A(3,1), 4, MPI_INT, pe, tag, MPI_COMM_WORLD, err);
```

This first index is the one contiguous in memory.

Array Ordering

- Once again, we will make this issue less critical from the MPI perspective once we learn about user defined datatypes in the advanced talk, but…

- this is extremely important to understand for general performance reasons. Notice how the C code iterates over the last (j) variable in the inner loop while the Fortran uses the first (i) in our serial code? This would kill our performance if it was the other way around. And I mean in the serial code. Go ahead and try…

```
for(i = 1; i <= ROWS; i++) {
     for(j = 1; j <= COLUMNS; j++) {
          Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                      Temperature_last[i][j+1] + Temperature_last[i][j-1]);
     }
}
```

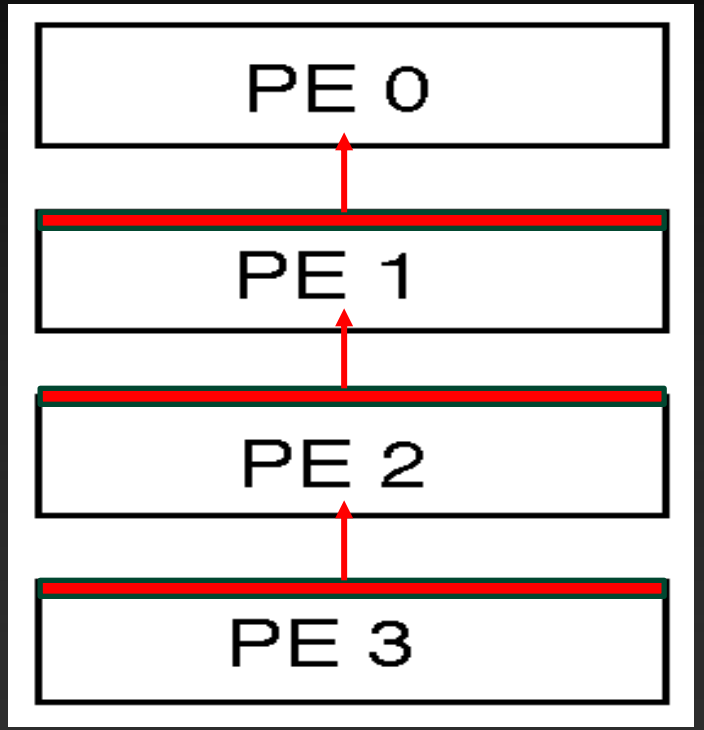
```
do j=1,columns
     do i=1,rows
          temperature(i,j)= 0.25 * (temperature_last(i+1,j)+temperature_last(i-1,j) + &
                                    temperature_last(i,j+1)+temperature_last(i,j-1) )
     enddo
enddo
```

Sending Multiple Elements

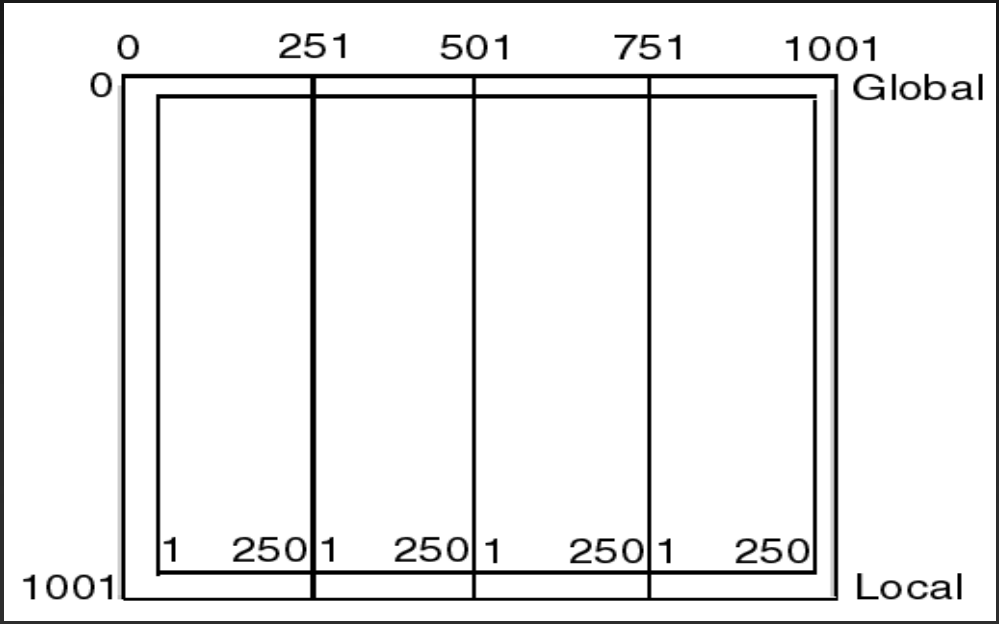
```
if ( mype != 0 ){
   up = mype - 1
   MPI_Send( t, COLUMNS, MPI_FLOAT, up, UP_TAG, comm);
}
```

Alternatively (and there are many ways to do this):

```
up = mype - 1
if ( mype == 0 ) up = MPI_PROC_NULL;
MPI_Send( t, COLUMNS, MPI_FLOAT, up, UP_TAG, comm);
```

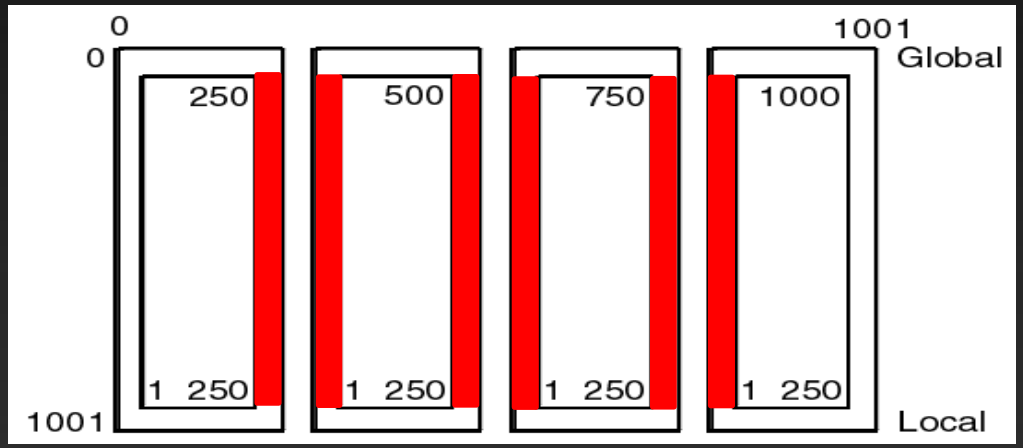


Simplest Decomposition for Fortran Code



Simplest Decomposition for Fortran Code

Then we send strips to ghost zones like this:



Same ghost cell structure as the C code, we have just swapped rows and columns.

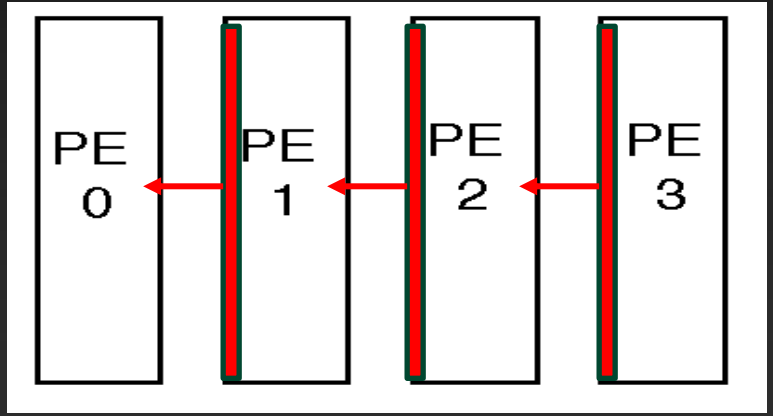
Sending Multiple Elements in Fortran

```
if( mype.ne.0 ) then
    left = mype - 1
    call MPI_Send( t, ROWS, MPI_REAL, left, L_TAG, comm, ierr)
endif
```

Alternatively (and there are many ways to do this):

```
left = mype - 1
if( mype.eq.0 ) left = MPI_PROC_NULL
call MPI_Send( t, ROWS, MPI_REAL, left, L_TAG, comm, ierr)
endif
```

Note: You may also `MPI_Recv from MPI_PROC_NULL`



Main Loop Structure

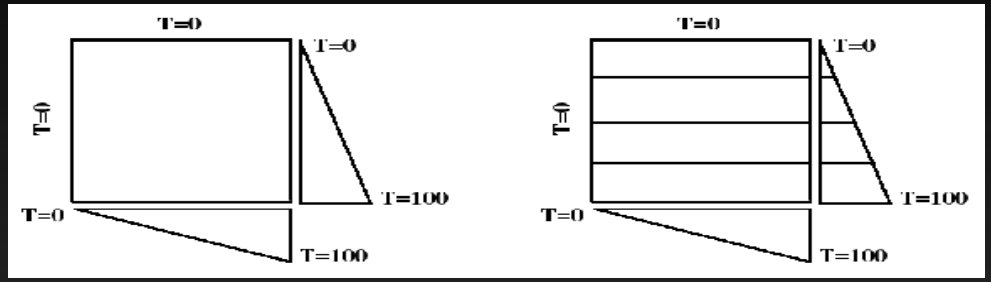
```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations )
                Calculate new Temperature
                Copy into temp array
                ...
```

Compute Phase
(almost unchanged)

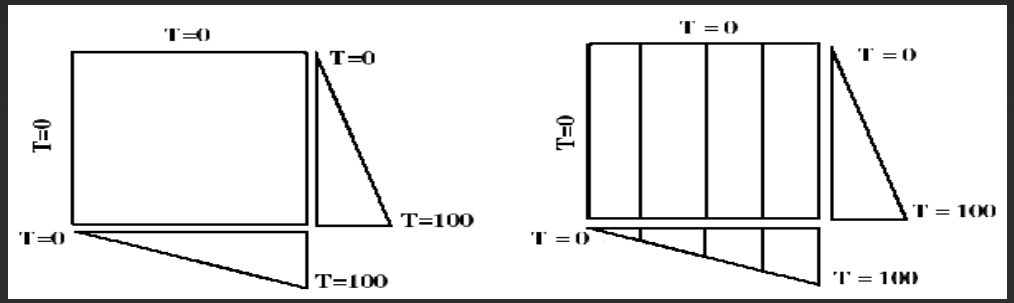
```
Send values down (Temperature or Temperature_last?)
Send values up
Receive values from above into ghost zone (Temperature or Temperature_last?)
Receive values from below into ghost zone

Find max change
Synchronize?
```

Communicate Phase
(all new)

Boundary Conditions



Both C and Fortran will need to set proper boundary conditions based upon the PE number.



Two ways to approach this exercise.

- Start from the serial code
- Start from the template (“hint”) code

Starting files in Exercises/MPI:

`laplace_serial.c`	`laplace_serial.f90`
`laplace_template.c`	`laplace_template.f90`

You can peek at my answer in /Solutions

`laplace_mpi.c`	`laplace_mpi.f90`

MPI Template for C

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

    int i, j;
    int max_iterations;
    int iteration=1;

    // the usual MPI startup routines
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    // verify only NPES PEs are being used
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    // PE 0 asks for input
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    // bcast max iterations to other PEs
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    if (my_PE_num==0) gettimeofday(&start_time,NULL);

    initialize(npes, my_PE_num);

    while ( dt_global > MAX_TEMP_ERROR && iteration <= max_iterations ) {

        // main calculation: average my four neighbors
        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                            Temperature_last[i][j+1] + Temperature_last[i][j-1]);
            }
        }

        // COMMUNICATION PHASE: send and receive ghost rows for next iteration
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

        dt = 0.0;
.
.
.
```

MPI Template for Fortran

```
.
.
program mpi
      implicit none
      include    'mpif.h'

      !Size of plate
      integer, parameter             :: columns_global=1000
      integer, parameter             :: rows=1000

      double precision, dimension(0:rows+1,0:columns+1) :: temperature, temperature_last

      !usual mpi startup routines
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      !It is nice to verify that proper number of PEs are running
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      !Only one PE should prompt user
      if( mype == 0 ) then
         print*, 'Maximum iterations [100-4000]?'
         read*,   max_iterations
      endif

      !Other PEs need to recieve this information
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      call cpu_time(start_time)

      call initialize(temperature_last, npes, mype)

      !do until global error is minimal or until maximum steps
      do while ( dt_global > max_temp_error .and. iteration <= max_iterations)

         do j=1,columns
            do i=1,rows
               temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                                      temperature_last(i,j+1)+temperature_last(i,j-1) )
            enddo
         enddo
```

Some ways you might go wrong…

You have two main data structures

- Temperature
- Temperature_last

Each has

- Boundary Conditions (unchanged through entire run)
- Ghost zones (changing every timestep)

Each iteration

- Copying/calculating Temperature to/from Temperature_last
- Sending/receiving into/from ghost zones and data

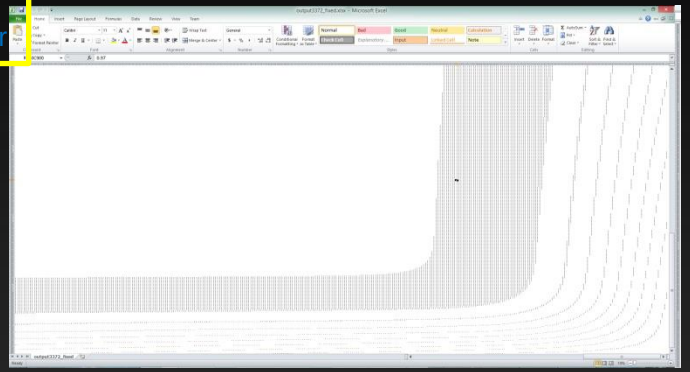
It is easy to mix these things up. I suggest you step through at least the initialization and first time step for each of the above combinations of elements.

There are multiple reasonable solutions. Each will deal with the above slightly differently.

How do you know you are correct?

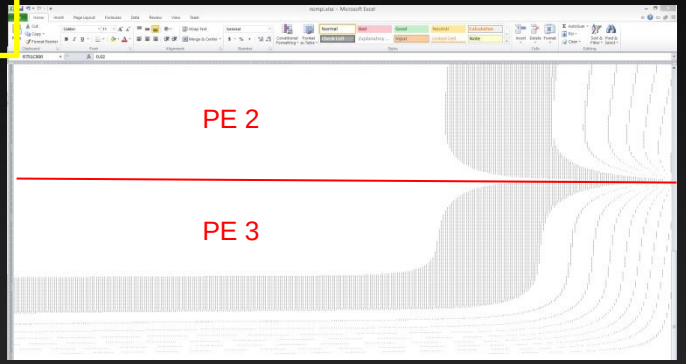
How do you know you are correct?

Bottom Right Corner



Working MPI Solution

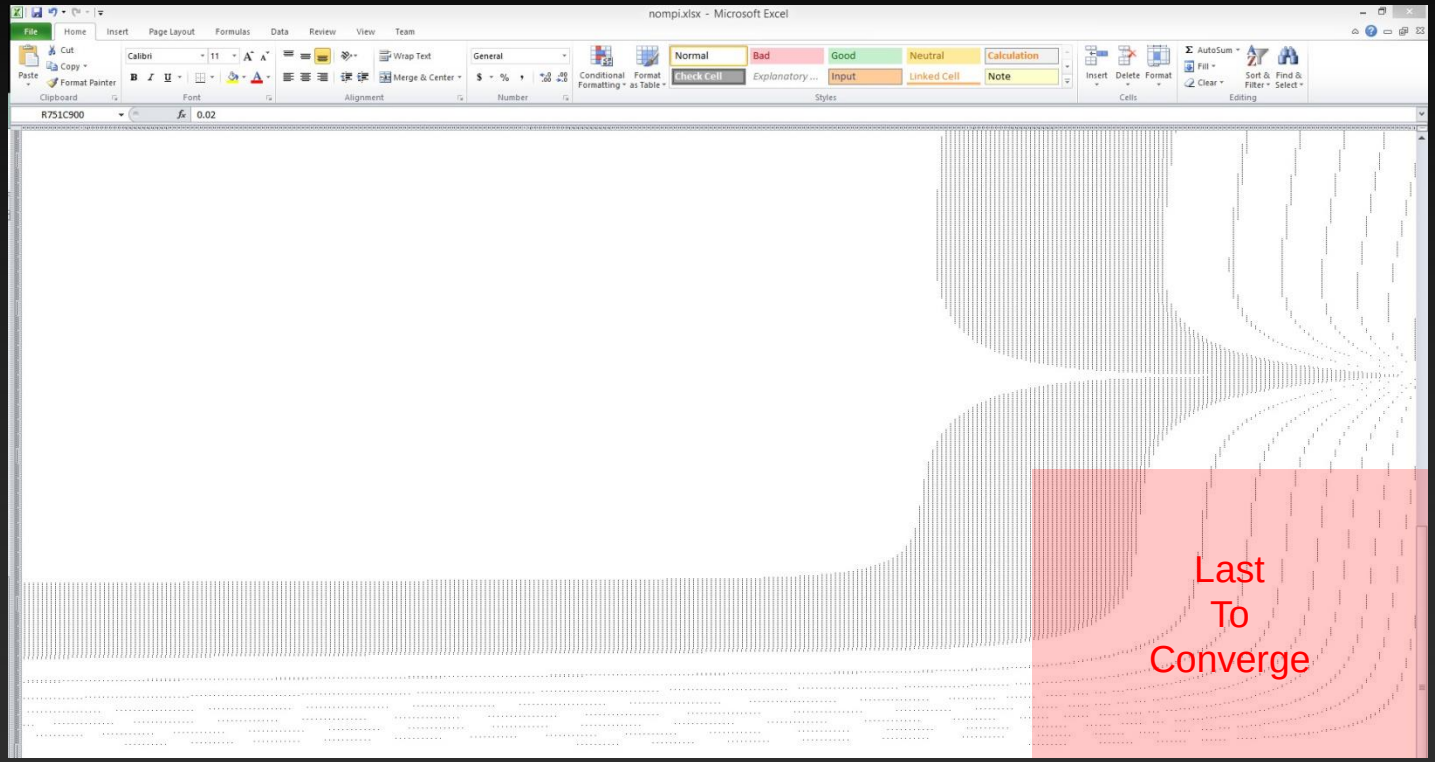
Bottom Right Corner



MPI Routines Disabled

Both converge at 3372 steps!

All the action is here.



Check for yourself.

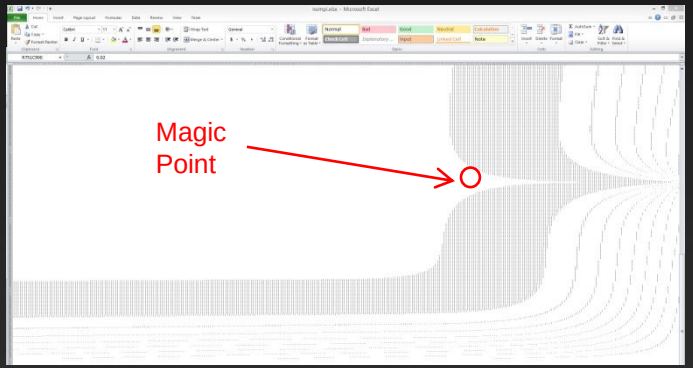
```
void output(int my_pe, int iteration) {

  FILE* fp;
  char filename[50];

  sprintf(filename,"output%d.txt",iteration);

  for (int pe = 0; pe<4; pe++){
    if (my_pe==pe){

      fp = fopen(filename, "a");

      for(int y = 1; y <= ROWS; y++){
        for(int x = 1; x <= COLUMNS; x ++){
          fprintf(fp, "%5.2f ",Temperature[y][x]);
        }
        fprintf(fp,"\n");
      }

      fflush(fp);
      fclose(fp);

    }
    MPI_Barrier(MPI_COMM_WORLD);

  }

}
```

C:

```
if (my_PE_num==2)
      printf("Global coord [750,900] is %f \n:", Temperature[250][900]);
```

Fortran:

```
if (mype==2) then
        print*, 'magic point', temperature(900,250)
endif
```



- Human Readable
- 1M entries
- Visualize. I used Excel (terrible idea).

- If about 1.0, probably good
- Otherwise (like 0.02 here) probably not

A Quick Note About Our Pace Before We Start

This exercise is doable during the exercise session time allotted if all goes well.

That has been our historical standard.

However some of you may need a little more time. That is fine.

We will not “spoil” the exercise during the Laplace Exercise Review tomorrow.

Those of you that need additional time can use your accounts through next week.

Please take advantage of that. If you complete this exercise, you understand MPI.

Laplace Exercise

1. You copied a directory called Exercises/MPI into your home directory. Go there and you will see the files:

   `laplace_template.c` and `laplace_serial.c`
   or
   `laplace_template.f90` and `laplace_serial.f90`

2. The templates are "hint" files with sections marked >>>>> in the source code where you might add statements so that the code will run on 4 PEs. You can start from either these or from the serial code, whichever you prefer. A useful Web reference for this exercise is the Message Passing Interface Standard at:

   http://www.mpich.org/static/docs/latest/

3. To compile the program as it becomes an MPI code, execute:

   ```
   mpicc laplace_your_mpi.c
   mpif90 laplace_your_mpi.f90
   ```

4. In an interactive session (with at least 4 Pes: "interact -n 4"), you can just run these as:

   ```
   mpirun -n 4 a.out
   ```

5. You can check your program against one possible solution in the Solutions directory:

   `laplace_mpi.c` or `laplace_mpi.f90`