Compiling

We will be using standard Fortran and C compilers on this Pathway. They should look familiar.

We will slightly prefer the NVIDIA compilers (the Intel or gcc or AMD or Clang ones would also be fine).

Note that on Delta you would normally have to enable this compiler with

```
module load nvhpc/24.1
```

I have put that in the .bashrc file that we will all use.

Configure Your Environment

Let’s get the boring stuff out of the way now. This will configure your environment by copying over the exercise files and setting up your .bashrc so that you can do our initial exercises, as well as exercises for the following two modules.

- Log on to Delta.
- Run the setup script that will copy over the Exercises directory we will all use. It will also automatically load the right compiler using your .bashrc script whenever you login. Note that linux shells are case sensitive.

```
/projects/becs/urbanic/Setup
```

- As told, logout and log back on again to complete the setup. You won’t need to do that in the future.

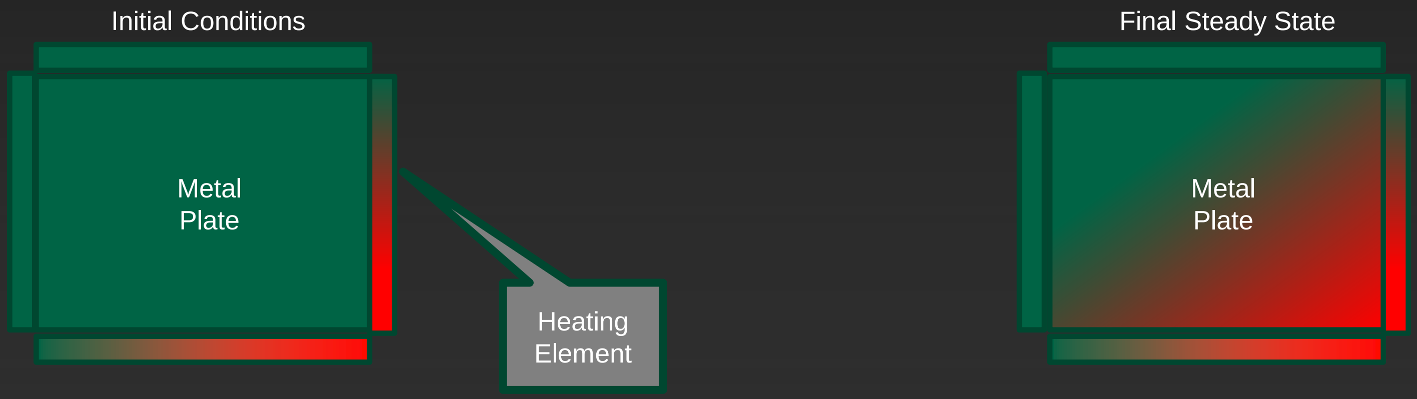
Our Files For This Pathway

After you run the setup script, you will have the following directories in your home directory. We will be using them in future modules.

```
/Exercises
       /Test
       /OpenMP
       /OpenACC
       /MPI
```

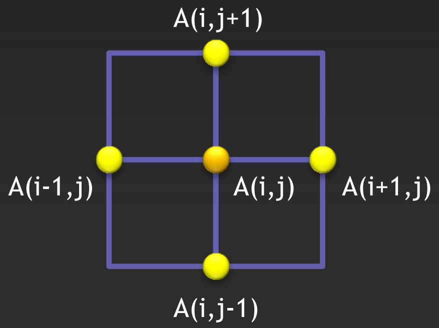
Our Foundation Exercise: Laplace Solver

- 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 C Code (kernel)

```c
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

PITTSBURGH SUPERCOMPUTING CENTER

Whole C Code

```c
#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");
}
```

Whole Fortran Code

```fortran
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
```

```fortran
! 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
```

Exercises For This Module.

Our first exercises will be to compile, run and time the Laplace code using the two programming models in this Pathway: OpenMP and MPI.

This will get you familiar with the programming environment in preparation for our actual parallel programming in the next two modules.

It will also allow you to experience some parallel scaling first hand.

Specifically, our goal will be to compile the Laplace code in OpenMP and run it on varying numbers of *threads* to see how much it speeds up.

We will do the same thing using MPI to run with multiple *processes*.

Don't worry if it seems like we are skipping over some details today - we are. But those details will be made clear in our following modules.

OpenMP (Exercise 1)

Go into the OpenMP folder inside the Exercises folder. You will see codes there called *laplace_omp.c* and *laplace_omp.f90*. Select whichever language most interests you.

To compile with OpenMP we do either

```
nvc -mp laplace_omp.c
```

or

```
nvfortran -mp laplace_omp.f90
```

Now we have an executable called *a.out*. But we need to ask for a compute node with multiple cores allocated in order to run. The Slurm command to get us a command line (*--pty bash*) on a compute node (*--nodes=1*) with 32 cores (*--cpus-per-task=32*) is:

```
srun --account=becs-delta-cpu --partition=cpu-interactive  --nodes=1 --cpus-per-task=32 --pty bash
```

*I am assuming the we all either have the same *--account*, or you know which one you should be using.

OpenMP

You will notice a few messages as Slurm find the resources, and then your command line will change to something with a compute node number in it.

From the command line on this compute node we can run up to 32 cores. OpenMP allows us to control core usage with the *OMP_NUM_THREADS* environment variable. You learned about these in the Intro to Delta module.

Set this variable to request 1 core (*export OMP_NUM_THREADS=1*) and run with *a.out* to find the baseline run time for the Laplace code. If you select 4000 iterations, the code will run to a complete solution. It reports its own runtime for you.

Now, try varying number of cores up to 32 and see what kind of speedup you experience. Note that you do not need to recompile the code. Just change the environment variable and run a.out. Record these times for our discussion.

exit the compute node when you are finished. You should find yourself returned to the login node.

MPI (Exercise 2)

Now we will do a similar exercise using MPI. Go into the MPI folder inside the Exercises folder. You will see codes there called *laplace_mpi.c* and *laplace_mpi.f*. Select whichever language most interests you.

To compile with MPI we do either

```
mpicc laplace_mpi.c
```

or

```
mpif90 laplace_mpi.f90
```

Again, you have an executable called *a.out*. Now you need to ask for a compute node with multiple processes allocated in order to run. Similar, but not identical, to the previous Slurm command, the one to get us a command line on a compute node with 4 processes (`--nodes=1 --tasks=4 --tasks-per-node=4`) is:

```
srun --account=becs-delta-cpu --partition=cpu-interactive  --nodes=1 --tasks=4 --tasks-per-node=4 --pty bash
```

MPI

With MPI we will limit ourselves to a single timing run, using 4 processes. The command to run our *a.out* executable on our four available processes is

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

Record this time for our later discussion.

Exit the compute node when you are finished.