APMA2822b Report Justin Miller
Transient Heat Eq. In Cylindrical Coordinates

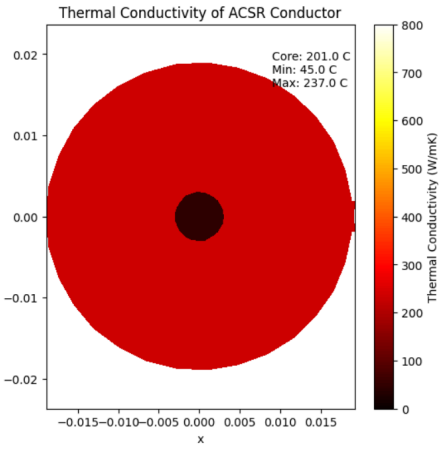
1. Introduction

a. Context

In this paper, I am exploring an application of High Performance Computing to my area of research which is in the power transmission industry. A pressing issue facing clean energy generation and electrification is the insufficient transmission infrastructure in the US and around the world. With increased load and congestion, there is the need to reevaluate and increase the safe current limits that existing transmission lines can carry. The biggest unknown in this is the varying wind speeds that impact convection rates and hence, wire temperature. This project will attempt to simulate the temperature dynamics within the wire over a period of increased loading. Once modeled, this project will optimize the simulation runtime to a practical duration by utilizing shared and distributed memory with GPUs.

b. Simulation Space

The physical system that will be simulated is a cross section of an ACSR Linnet transmission wire. This conductor has a radius of 1.8 cm and is composed of an outer layer of aluminum with a smaller steel core for structural reinforcement. The stranding of the conductor can be modeled by a solid cylinder with an adjusted thermal conductivity coefficient. The situation being simulated will be an emergency loading situation. This occurs when part of the grid is forced online and utility operators are forced to route power through other lines. When this occurs, our line of interest heats up over the course of a couple of hours. Knowing how this line heats up can provide critical information to prevent sag violations and annealing inside the core of the wire.



This problem requires much more time resolution than spatial resolution. Grid operators are more concerned with second-by-second line temperature over the full time period than they are with the small-scale spatial temperature differences within the metal. Note that this is a key factor in the solver method selected (see section 2 below).

2. Problem Formulation

a. Governing Equation

The transient heat transfer system is governed by the Fourier-Biot Equation. For this application the equation is in the cylindrical coordinates with the dz/dt term omitted due to negligent heat transfer along the axis of the wire. This leads to the equation shown below:

$$\frac{1}{r}\frac{\partial}{\partial r}\left(kr\frac{\partial T}{\partial r}\right)+\frac{1}{r^2}\frac{\partial T}{\partial \phi}\left(k\frac{\partial T}{\partial \phi}\right)+\dot{e}_{\text{gen}}=\rho c\frac{\partial T}{\partial t}$$

The constants k, p and c are known material properties that can be input into the model for each cell as a function of r (to account for different metals in the wire). The energy generation term $\boldsymbol{e_{gen}}$ (also referred to as q_{int}) is the internal heat generated resulting from the current load (I) through each cell. This current distribution is determined in adherence to the skin effect in AC transmission systems and is precalculated into a lookup table at the start of each simulation run.

b. Description of the Finite Difference Method

To implement this equation in a solver, an implicit Finite Difference scheme is used. This method is used instead of a more traditional time stepping method for a couple of reasons.

The main factor to use an implicit scheme (contrary to an explicit scheme) is the fact that our problem application requires much more time resolution than spatial resolution. An explicit scheme is not easily parallelizable over time (due to dependence on having solved previous time steps), and since most of our computation is propagating in the time direction we benefit from parallelization much more with an implicit scheme.

Additionally, an implicit scheme using the backwards difference method for the time dimension guarantees us unconditional stability. An explicit scheme would require a time step to be sufficiently small compared to the spatial sizing to satisfy the CFL condition for stability. That would cause difficulties for this given problem given both the long simulation duration and the small cell size created near the singularity.

c. Overview of the Implicit Time-Stepping Scheme

To discretize the equation, the second-order finite difference approximation is used in the spatial dimensions (r and θ) and the second-order finite difference approximation is also used in the temporal dimension.

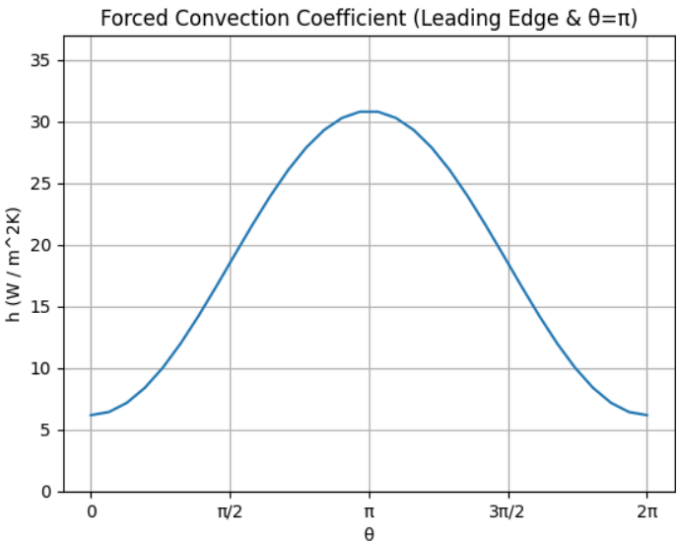
d. Problem Setup and Domain

To solve this problem, a 3D Space-Time grid is created with the dimensions (r, θ, t). For grid sizing, the Δr and Δθ are determined by splitting the radius length (1.8 cm) and the θ range (2π). The total time length of the simulation (t) and the step size of time (Δt) are adjustable depending on the simulation requirements. To best simulate the needs of the emergency loading situation, for the rest of this project the number of cells in each direction will be (32, 64, 512) and Δt will be 10 seconds.

e. Description of the boundary and initial conditions

The initial condition is a constant matching the ambient temperature (here, 25 C). This is to simulate an unloaded metal conductor and is initialized for the first two timestamps to allow for the second order temporal FD approximation. The first boundary condition is the periodic boundary condition. This is simply enforced by ensuring that the last element in the θ direction loops around to the first element.

The second boundary condition is the forced convection condition. The Finite Difference appx. for this is enforced whenever we are at the outermost cell in direction r. The plot on the right shows the convection coefficient as a function of θ, where θ=π is the leading edge of the wire in cross flow. This distribution results from the impact of the wake on forced convection in a constant wind, resulting in an asymmetry which makes this problem require finite difference to solve.



3. Sequential Implementation

To start exploring this problem, the solver was implemented sequentially in both Python and C++. A single iterative step to solve implicitly is shown below (the C++ version).

```
// Main Loop
for (int i = 1; i < N_r -1; ++i){
    for (int j = 0; j < N_theta; ++j){

        // Adjusting for periodic boundary condition
        j_plus = j + 1;
        j_minus = j - 1;
        if (j == 0){
            j_minus = N_theta - 1;
        }
        if (j == N_theta - 1){
            j_plus = 0;
        }

        // Continuing loop (through time index)
        for (int k = 2; k < N_t + 2; ++k){

            // Compute the temperature using the finite difference equations
            T_new[(k * N_r * N_theta) + (j * N_r) + i] = (1 / ((4 * r[i]*r[i] * delta_theta*delta_theta + 4 * delta_r*delta_r) + \
                        ((3 * r[i]*r[i] * delta_r*delta_r * delta_theta*delta_theta) / (alpha(r[i], theta[j]) * delta_t)))) * ( \
                        (2 * r[i]*r[i] * (delta_r*delta_r) * (delta_theta*delta_theta) * q_int(r[i], theta[j], curr_lookup[i]) / km(r[i], theta[j])) + \
                        (T[(k * N_r * N_theta) + (j * N_r) + i + 1] * (2 * r[i]*r[i] * (delta_theta*delta_theta) + r[i] * delta_r * (delta_theta*delta_theta))) + \
                        (T[(k * N_r * N_theta) + (j * N_r) + i - 1] * (2 * r[i]*r[i] * (delta_theta*delta_theta) - r[i] * delta_r * (delta_theta*delta_theta))) + \
                        ((T[(k * N_r * N_theta) + ((j_plus) * N_r) + i] + T[(k * N_r * N_theta) + ((j_minus) * N_r) + i]) * (2 * (delta_r*delta_r))) + \
                        (((r[i]*r[i] * delta_r*delta_r * delta_theta*delta_theta) / (alpha(r[i], theta[j]) * delta_t)) * \
                        (4 * T[((k - 1) * N_r * N_theta) + (j * N_r) + i] - T[((k - 2) * N_r * N_theta) + (j * N_r) + i]))    );


        }
    }
}
```

This snippet makes calls to the global functions alpha(), km() and q_int() which return the thermal diffusivity, the thermal conductivity (k), and the internal heat generation of any given cell. In addition, the convective boundary condition is applied to all thetas at R = N_r - 1.

```
// Apply convective boundary conditions
for (int j = 0; j < N_theta; ++j){
    for (int k = 2; k < N_t + 2; ++k){
        T_new[(k * N_r * N_theta) + (j * N_r) + (N_r - 1)] = (km(R, M_PI * 2) * T_new[(k * N_r * N_theta) + \
                                                             (j * N_r) + (N_r - 2)] + h(R, theta[j]) * delta_r * T_inf) / \
                                                             (km(R, M_PI * 2) + h(R, theta[j]) * delta_r);  // Convective boundary
    }
}
```

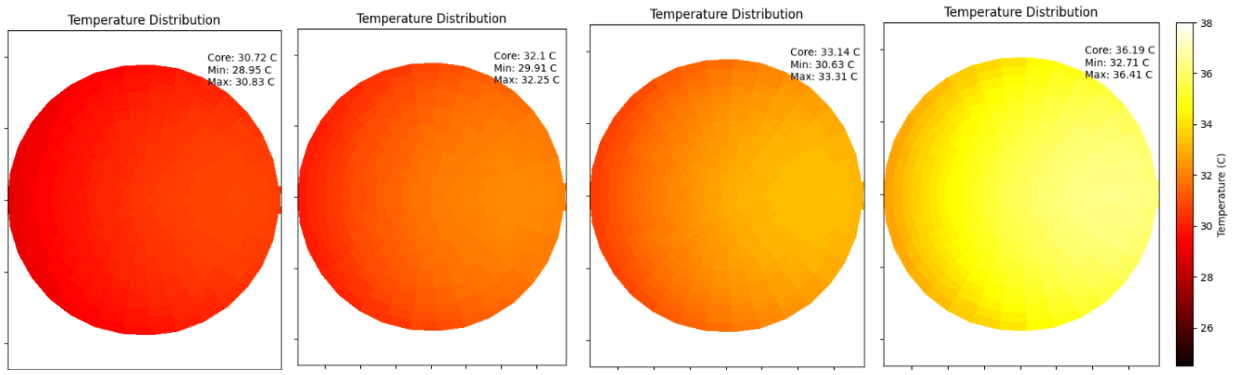
Lastly, a L1 convergence check is run and will break the program if it exceeds tolerance of 1e-6.

a. Addressing Singularities in the Sequential Implementation

Because of the periodic nature in the θ direction, this implementation has a singularity at r=0. In order for the simulation to converge it is important to handle these cells separately to maintain the heat flux from the cells at r=1. To do this, the solver treats the cells at r=0 as having no size and calculates the heat transfer between the neighboring cells at (r, θ, :) and the cell directly across the singularity at (r, $\theta_{opposite}$, :). This keeps the flow of heat continuous, and it is shown to be successful in visualizations of the simulation. The one caveat is that there must be an assertion to keep the number of cells in the theta direction divisible by 2. The FD equation is the same as the above snippet with the exception that *T[i, j, k]* is replaced by *T[i, (j-N_theta/2)%N_theta, k]* to allow for the singularity spanning heat conduction.

b. Challenges Encountered and Results

The main challenge implementing this sequential version was deriving and coding the finite difference approximation. Given all the inputs, this turned into a complicated code snippet (as shown above) and took a fair amount of debugging to get right. Once completed, though, it was fairly straightforward to test and visualize. The final result aligned with what is expected from existing lumped capacitance models (for transient heat convection/conduction systems) and the conductive behavior over time was smooth throughout the domain (can't include the animation from the presentation in this report, so here are a few frames of that below for reference).



4. Shared Memory Model

a. Implementation

To parallelize the above Finite Difference scheme using a GPU, there are a few key parts. The first is memory initialization and management. For this implementation, it is necessary to allocate and copy the two versions of the temperature solution grid as well as a variable to record local convergence error and another variable to hold data from the current distribution input. The allocation of these variables, as well as the initialization of our GPU grid & blocks are shown in the snippet below.

```
double overall_max_diff = 9999.9;
double *d_T, *d_T_new, *d_local_max_diff, *d_curr_lookup;

// Setting up cuda error code
cudaError_t GPU_ERROR;

// Allocating mem: be concious of the initial condition of length 2 that needs to be added to N_t
GPU_ERROR = cudaMalloc( (void**) &d_T, N_r * N_theta * (N_t + 2) * sizeof(double));
GPU_ERROR = cudaMalloc( (void**) &d_T_new, N_r * N_theta * (N_t + 2) * sizeof(double));
GPU_ERROR = cudaMalloc( (void**) &d_local_max_diff, N_r * N_theta * (N_t + 2)* sizeof(double));
GPU_ERROR = cudaMalloc( (void**) &d_curr_lookup, N_r * sizeof(double));

// Define block and grid sizes
dim3 blockSize(TILE_SIZE_X, TILE_SIZE_Y, TILE_SIZE_Z);
dim3 gridSize((N_r + TILE_SIZE_X - 1) / TILE_SIZE_X, (N_theta + TILE_SIZE_Y - 1) / TILE_SIZE_Y,  ((N_t + 2) + TILE_SIZE_Z - 1) / TILE_SIZE_Z);

// Populating initial diff array
double *local_max_diff = new double[N_r * N_theta * N_t];
for (int i = 0; i < N_r; ++i) {
    for (int j = 0; j < N_theta; ++j) {
        for (int k = 0; k < (N_t + 2); ++k)
            local_max_diff[(k * N_theta * N_r) + (j * N_r) + i] = 0.0;
    }
}

// Copy data from host to device
GPU_ERROR = cudaMemcpy(d_T, T, N_r * N_theta * (N_t + 2) * sizeof(double), cudaMemcpyHostToDevice);
GPU_ERROR = cudaMemcpy(d_T_new, T_new, N_r * N_theta * (N_t + 2) * sizeof(double), cudaMemcpyHostToDevice);
GPU_ERROR = cudaMemcpy(d_local_max_diff, local_max_diff, N_r * N_theta * (N_t + 2) * sizeof(double), cudaMemcpyHostToDevice);
GPU_ERROR = cudaMemcpy(d_curr_lookup, curr_lookup, N_r * sizeof(double), cudaMemcpyHostToDevice);
```

The tile sizes (x, y, z) were set to correspond roughly to the (r, θ, t) sizes. The fastest setting found to break down these 3D tiles was to set them as 32, 8, and 4, respectively. Once initialized, the next step was to run the iteration scheme and execute the kernel at each step. This kernel, designed to encompass the FD scheme, singularity handling and boundary conditions is below.

```
__global__ void finiteDifferenceKernel(double *T, double *T_new, int N_r, int N_theta, int N_t, double delta_r, double delta_theta, double delta_t, double *local_max_diff, double* curr_lookup){
    // Defining thread indices
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // x-index (grid point in r direction)
    int j = blockIdx.y * blockDim.y + threadIdx.y;  // y-index (grid point in theta direction)
    int k = blockIdx.z * blockDim.z + threadIdx.z;  // z-index (grid point in t direction)

    // Helper tmp vars
    double sum_over_theta;
    double singularity_mean;
    double error;
    int j_plus;
    int j_minus;

    // Adjusting for periodict boundary condition
    j_plus = j + 1;
    j_minus = j - 1;
    if (j == 0){
        j_minus = N_theta - 1;
    }
    if (j == N_theta - 1){
        j_plus = 0;
    }

    // Edge case for center point
    if (i == 0 && j >= 0 && j < N_theta && k>=2 && k < (N_t + 2)){

    // Compute the temperature using the finite difference equations with r adjusted to cross singularity
    T_new[(k * N_r * N_theta) + (j * N_r) + i] = (1 / ((4 * delta_r*(i + 1)*delta_r*(i + 1) * delta_theta*delta_theta + 4 * delta_r*delta_r) + ((3 * delta_r*(i + 1)*delta_r*(i + 1) * delta_r*delta_r * delta_th
                (2 * delta_r*(i + 1)*delta_r*(i + 1) * (delta_r*delta_r) * (delta_theta*delta_theta) * q_int(delta_r*(i + 1), delta_theta*j, curr_lookup[i]) / km(delta_r*(i + 1), delta_theta*j)) + \
                (T[(k * N_r * N_theta) + (j * N_r) + i + 1] * (2 * delta_r*(i + 1)*delta_r*(i + 1) * (delta_theta*delta_theta) + delta_r*(i + 1) * delta_r * (delta_theta*delta_theta))) + \
                (T[(k * N_r * N_theta) + ((int(((j - N_theta /2)) % N_theta)) * N_r) + i + 1] * (2 * delta_r*(i + 1)*delta_r*(i + 1) * (delta_theta*delta_theta) - delta_r*(i + 1) * delta_r * (delta_theta*d
                ((T[(k * N_r * N_theta) + ((j_plus) * N_r) + i] + T[(k * N_r * N_theta) + ((j_minus) * N_r) + i]) * (2 * (delta_r*delta_r))) + \
                (((delta_r*(i + 1)*delta_r*(i + 1) * delta_r*delta_r * delta_theta*delta_theta) / (alpha(delta_r*(i + 1), delta_theta*j) * delta_t)) * (4 * T[((k - 1) * N_r * N_theta) + (j * N_r) + i] - T[
    }
```

```
    // Main Loop (run for everything else in the correct bounds)
    if (i > 0 && i < (N_r - 1) && j >= 0 && j < N_theta && k >= 2 && k < (N_t + 2)) {


    // Compute the temperature using the finite difference equations
    T_new[(k * N_r * N_theta) + (j * N_r) + i] = (1 / ((4 * (delta_r*i)*(delta_r*i) * delta_theta*delta_theta + 4 * delta_r*delta_r) + ((3 * (delta_r*i)*(delta_r*i) * delta_r*delta_r * delta_theta*delta_theta) / (alpha((delta_r*i
                (2 * (delta_r*i)*(delta_r*i) * (delta_r*delta_r) * (delta_theta*delta_theta) * q_int((delta_r*i), delta_theta*j, curr_lookup[i]) / km((delta_r*i), (delta_theta*j))) + \
                (T[(k * N_r * N_theta) + (j * N_r) + i + 1] * (2 * (delta_r*i)*(delta_r*i) * (delta_theta*delta_theta) + (delta_r*i) * delta_r * (delta_theta*delta_theta))) + \
                (T[(k * N_r * N_theta) + (j * N_r) + i - 1] * (2 * (delta_r*i)*(delta_r*i) * (delta_theta*delta_theta) - (delta_r*i) * delta_r * (delta_theta*delta_theta))) + \
                ((T[(k * N_r * N_theta) + ((j_plus) * N_r) + i] + T[(k * N_r * N_theta) + ((j_minus) * N_r) + i]) * (2 * (delta_r*delta_r))) + \
                ((((delta_r*i)*(delta_r*i) * delta_r*delta_r * delta_theta*delta_theta) / (alpha((delta_r*i), (delta_theta*j)) * delta_t)) * (4 * T[((k - 1) * N_r * N_theta) + (j * N_r) + i] - T[((k - 2) * N_r * N_theta) + (j * N

    }

    // Apply convective boundary conditions if we are at the edge
    if (i == (N_r - 1) && j >= 0 && j < N_theta && k >= 2 && k < (N_t + 2)){

        T_new[(k * N_r * N_theta) + (j * N_r) + (N_r - 1)] = (km(R, M_PI * 2) * T_new[(k * N_r * N_theta) + (j * N_r) + (N_r - 2)] + h(R, (delta_theta*j)) * delta_r * T_inf) / (km(R, M_PI * 2) + h(R, (delta_theta*j)) * delta_r);
    }

    // Populate field for convergence (e.g., check change in T - note excludes singularity error)
    local_max_diff[(k * N_r * N_theta) + (j * N_r) + i] = std::fabs(T_new[(k * N_r * N_theta) + (j * N_r) + i] - T[(k * N_r * N_theta) + (j * N_r) + i]);

    // Syncing
    __syncthreads();
}
```

Note that the main equation and the equation at the singularity are exactly the same as the sequential version. The same is true for the periodic and convective boundary conditions. The main difference here is separating these steps of the iteration into if blocks to run given the global coordinates currently being executed (i, j, k).

The last part of executing this on the GPU is the handling of the convergence condition. To do this, the maximum value of the device's local max diff array is found and compared to the experiment tolerance value. Note that everything must be synchronized before running this process. Also note that this process requires an expensive call to cudaMemcpy() to transfer the convergence criteria off of the device, so it is only called every 1000 iterations. Note that this can be adjusted to account for the expected iterations required to converge. This condition, as well as the calling of our final difference kernel in the iterative loop, is shown in the snippet below.

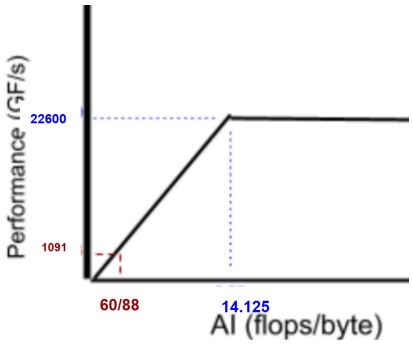
```cpp
// Perform finite difference iterations
for (int iter = 0; iter < max_iter; ++iter) {

    // Launch the kernel for this time step
    finiteDifferenceKernel<<<gridSize, blockSize>>>(d_T, d_T_new, N_r, N_theta, N_t, delta_r, delta_theta, delta_t, d_local_max_diff, d_curr_lookup);
    GPU_ERROR = hipGetLastError();
    if (GPU_ERROR != 0) {
        std::cerr << "CUDA kernel launch failed: " << hipGetErrorString(GPU_ERROR) <<  std::endl;
    }
    GPU_ERROR = cudaDeviceSynchronize();


    // Tolerance check step (for now, do every 100)
    if (iter % 1000 == 999){


        // Copy d_local_max_diff
        GPU_ERROR = cudaMemcpy(local_max_diff, d_local_max_diff, N_r * N_theta * (N_t + 2) * sizeof(double), cudaMemcpyDeviceToHost);


        // Getting max of local_max_diff
        overall_max_diff = 0.0;
        for (int i = 0; i < N_r; ++i) {
            for (int j = 0; j < N_theta; ++j){
                for (int k = 2; k < N_t + 2; ++k)
                overall_max_diff = std::max(overall_max_diff, local_max_diff[(k * N_theta * N_r) + (j * N_r) + i]);
            }
        }

        std::cout << "Iteration " << iter << ": Max Error: " << overall_max_diff << std::endl;

        if (overall_max_diff < tol){
            std::cout << "Exited at iteration: " << iter << std::endl;
            break;
        }
    }

    // Swap the pointers (T and T_new are swapped for the next iteration)
    double *temp = d_T;
    d_T = d_T_new;
    d_T_new = temp;
}

// Copy the result back to the host
GPU_ERROR = cudaMemcpy(T_new, d_T_new, N_r * N_theta * (N_t + 2) * sizeof(double), cudaMemcpyDeviceToHost);
```

b. Performance

The maximum performance achieved by the shared memory model is given in the table below.

Tolerance	Runtime (s)	Iterations	Time / Iteration (µs)	Bytes / Iteration	Gb/s	FLOP / Iteration	Gf/s
1e-6	3.946	37,000	107	92,274,688	862.380	62,914,560	587.986

This achieves roughly a 54% theoretical efficiency for both bandwidth and flops. The sketched roofline model to the right shows how the memory-limited nature of the problem impacts the overall performance. While not nearing the theoretical value, this runtime improvement is practical success, more on that will be discussed later when comparing against all created models.



a

c. Note on Optimization

For the shared memory model, there was some tuning that could be done to get the best possible performance. The simplest was the built in compiler optimization: when running hipcc with the -O2 argument the performance increased from 4.37 seconds to 3.93 seconds, a 10% improvement. The other improvement made was in the thread count split between the three dimensions. A variety of combinations that resulted in the max thread count were tried, and the resulting decision is shown in the table below. Note (x,y,z) corresponds to the (r, θ, t) axes.

Thread Pattern	**Runtime (s)**	**Thread Pattern**	**Runtime (s)**
(2, 2, 256)	9.027	**(8, 8, 16)**	4.457
(4, 4, 64)	4.952	**(32, 8, 4)**	3.946

5. Distributed Memory Model

The second parallelization strategy used the distributed memory model offloading computation to 4 individual GPUs. This method of splitting up our problem and assigning calculations to independent devices is especially useful for large problems where the size of the domain outweighs the memory available in a single device. In this case, success with the distributed memory model will be determined by any runtime improvements from the single device implementation.

The first relevant code snippet is the splitting of the domain and assignment of computation. For this problem's uneven domain, the most efficient way to split this problem is in the time domain. In addition to requiring the smallest possible amount of data for halo exchange, this is also the most straightforward to implement, only requiring a split in one dimension. For the scope of this project, 4 host CPUs are used with 1 MPI task each to split the problem to a corresponding GPU. See the snippet below for rank and GPU assignment.

```
int rankID = 0;
int num_ranks = 1;

MPI_Init(&argc, &argv);  // Initialize MPI
MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);  // Get number of ranks
MPI_Comm_rank(MPI_COMM_WORLD, &rankID);  // Get current rank

// Decompose the domain across ranks... doing it in the time dimension only
int chunk = N_t / num_ranks; // Chunk size per rank
int remainder = N_t % chunk * num_ranks;
int nlocal = chunk;
if (rankID < remainder % chunk) nlocal++;  // Adjust for uneven chunk sizes

int N_start_t = (rankID) * nlocal + 2;  // Adding 2 spots for boundary conditions
int N_end_t = (rankID + 1) * nlocal + 2;
```

```
// Setting up cuda to assign 1 to each rank
cudaError_t GPU_ERROR;

//check if GPUs are available... if not exit
int ndevices=0;
GPU_ERROR = cudaGetDeviceCount(&ndevices);
if (ndevices > 0){
  printf("%d GPUs have been detected\n",ndevices);
  GPU_ERROR = cudaSetDevice(rankID); //use device with ID 0
  printf("Assigning GPU %d\n", rankID);
}
else {
  printf("no GPUs have been detected, exiting\n");
  return 0;
}
```

Once the problem is split, we are effectively running the shared memory model iteration on each individual GPU. As such, the same exact kernel and parameters from the previous model is used, and the snippets from that section can be referenced above. The main difference created by using

4 GPUs instead of 1 is the requirement for an MPI reduction to calculate the convergence error. This reduction call is shown in the snippet below, where local_overall_max_diff is the individual GPU convergence error calculated as shown in the shared memory section above.

```
double global_max_diff;
MPI_Allreduce(&local_overall_max_diff, &global_max_diff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD)

// Breaking if we have reached global convergence condition
if (local_overall_max_diff < tol) {
    std::cout << "Converged after " << iter << " iterations." << std::endl;
break;
}
```

The one remaining differentiator is the requirement for halo exchange. For this implicit problem that has been split in the time direction, halo exchange is only required in one direction (sending to the right), although it still requires sending 2 rows in the time direction: t-1 and t-2. (recall the backwards difference scheme from section 1). This exchange snippet is shown below:

```
// Non-blocking MPI communication for boundary exchange (only need 1 for backwards difference)
MPI_Request send_request;
MPI_Request recv_request;

// Pulling down u from device
GPU_ERROR = cudaMemcpy(T_new, d_T_new, N_r * N_theta * (N_t + 2) * sizeof(double), cudaMemcpyDeviceToHost);
std::cout << "Recieved T_new on host " << rankID << std::endl;
GPU_ERROR = cudaDeviceSynchronize();

// Send data to the right, recieve from the left.
if (rankID < num_ranks - 1) {

    // Filling in buffer
    for (int i = 0; i < N_r; ++i){
        for (int j = 0; j < N_theta; ++j){
            send_buffer[(j * N_r) + i] = T_new[((N_end_t - 1) * N_r * N_theta) + (j * N_r) + i];
            send_buffer[(N_r * N_theta) + (j * N_r) + i] = T_new[((N_end_t - 2) * N_r * N_theta) + (j * N_r) + i];

        }
    }
    MPI_Isend(&send_buffer[0], N_r * N_theta * 2, MPI_DOUBLE, rankID + 1, 0, MPI_COMM_WORLD, &send_request);
    MPI_Wait(&send_request, MPI_STATUS_IGNORE);

}

if (rankID > 0){

    MPI_Irecv(&recv_buffer[0], N_r * N_theta * 2, MPI_DOUBLE, rankID - 1, 0, MPI_COMM_WORLD, &recv_request);
    MPI_Wait(&recv_request, MPI_STATUS_IGNORE);

    // Unpacking buffer
    for (int i = 0; i < N_r; ++i){
        for (int j = 0; j < N_theta; ++j){
            T_new[((N_start_t - 1) * N_r * N_theta) + (j * N_r) + i] = recv_buffer[(j * N_r) + i];
            T_new[((N_start_t - 2) * N_r * N_theta) + (j * N_r) + i] = recv_buffer[(N_r * N_theta) + (j * N_r) + i];

        }
    }
}

// After sending back T_new to device (for all ranks)
GPU_ERROR = cudaMemcpy(T_new, d_T, N_r * N_theta * (N_t + 2) * sizeof(double), cudaMemcpyDeviceToHost);
```

Note that for this run the halo exchange is run every 1000 iterations. This is done to limit the calls to the expensive memcpy(). While this is not the best practice, for this particular problem the time dependent behavior is very slow (material takes a while to heat up) so there is an overall

runtime improvement when forgoing some halo exchange. The chart below displays this runtime quirk by comparing time until convergence for different reduced halo transfer options.

Run Halo Every:	1000 iterations	100 iterations	10 iterations
Time until Convergence:	3.303 seconds	3.960 seconds	8.514 seconds

6. Conclusion

Overall, this project was a practical success, creating a correct model of transient heat transfer for a cylindrical object and significantly reducing the runtime to a convenient length. The final summary of improvement is shown in the table below.

	Time / Iteration (µs)	**Evaluated Runtime (s)**
Python Prototype	94,391	65 Min.
C++ Sequential	40,918	28 Min.
C++ GPU Shared	107	3.95 Sec.
C++ GPU Offloaded (4 GPUs)	89	3.30 Sec

There were a few significant issues worth mentioning in both the model and the parallelization. The largest issue was with the joule heating initialization ($\mathbf{q_{int}}$**).** The complexity here comes from the skin effect, which describes the exponential behavior of current distribution within a wire under AC load. When initializing this on a discretized grid I was getting some abnormally large values that caused convergence failures in the simulation. I was able to get it working for most current input and cell sizings, but there is still an unsolved bug that will sometimes cause failures and requires care when setting the input current. The other issue with this project came from time prioritization. The model itself took lots of time to implement and debug, restricting the amount of optimizations that I could try (such as CUDA streams).