CSC-201 Project 02

Write C program file implementing Heat 3D algorithm with DOALL parallelism in OpenMP. Compare the performance with sequential ordering.

Heat 3D Algorithm

The heat 3D algorithm, often used in scientific and engineering applications, solves the **heat equation** for a three-dimensional object. The heat equation is a partial differential equation (PDE) that describes how heat diffuses through a medium over time. To understand this algorithm, let's break down the concept step-by-step.

1. Basics of Heat Diffusion

In the context of 3D heat diffusion, imagine a cube (or another 3D object) where different points may have different temperatures initially. The heat will naturally flow from hot regions to cold regions, and over time, the temperature in the cube will become more uniform. The **heat equation** models this flow mathematically.

In 3D, the heat equation is written as:

$$\frac{\partial T}{\partial t} = \alpha(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2})$$

where:

- $T(x, y, z, t)$is the temperature at a point (x, y, z) and time t.
- α is the thermal diffusivity of the material, which tells us how quickly heat spreads.
- The terms $\frac{\partial^2 T}{\partial x^2},\ \frac{\partial^2 T}{\partial y^2},\ and\ \frac{\partial^2 T}{\partial z^2}$ are **second-order spatial derivatives**, which represent the rate of change of temperature in each direction.

2. Discretizing the Space (Finite Difference Method)

To solve this equation numerically on a computer, we break down space into a **3D grid** of points, where we calculate the temperature at each grid point. This process is called **discretization**.

- Assume the 3D object is divided into small cubes (grid cells) along the x, y, and z directions.
- Each point in this 3D grid has coordinates (i, j, k), where i, j, and k are indices representing the position along the x, y, and z directions, respectively.

3. Approximating the Derivatives

In a numerical approach, we approximate the spatial and time derivatives using finite differences. For the second spatial derivatives, we use the following finite difference approximation:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{\Delta x^2}$$

where:

- $T_{i+1,j,k},\ T_{i,j,k},\ and\ T_{i-1,j,k}$ are the temperatures at neighboring grid points along the x-axis.
- Δx is the distance between neighboring points along x.

Similar expressions are used for the y and z directions.

4. Time Stepping (Forward Euler Method)

To simulate how heat spreads over time, we step forward in time incrementally using **time steps** of size Δt. At each time step, we update the temperature at each grid point based on the heat equation.

The temperature at the next time step can be calculated as:

$$T_{i,j,k}^{new} = T_{i,j,k}^{old} + \Delta t \cdot \alpha(\frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{\Delta x^2} + \frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{\Delta y^2} + \frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{\Delta z^2})$$

This formula uses the **Forward Euler method**, a simple numerical method for stepping forward in time.

5. Boundary Conditions

The boundaries of the 3D object may have specific conditions (like holding a constant temperature or being insulated). These **boundary conditions** affect how heat flows at the edges of the object.

Common boundary conditions include:

- **Dirichlet boundary condition**: Specifying a fixed temperature at the boundaries.
- **Neumann boundary condition**: Specifying that there is no heat flow across the boundaries (insulated boundaries).

6. Algorithm Summary

The heat 3D algorithm proceeds as follows:

1. **Initialize the grid**: Set up the 3D grid with initial temperatures at each point.
2. **Set boundary conditions**: Define what happens at the edges of the grid.
3. **Iterate over time steps**:
    - For each grid point (i, j, k):
        - Calculate the new temperature based on the heat equation and the temperatures at neighboring points.
    - Update the temperature at each point after each time step.
4. **Stop the simulation** once a specified time is reached or the temperature distribution stabilizes.

7. Example Code

A simple pseudocode outline for the 3D heat algorithm:

```
alpha=0.01
dx,dy,dz=0.1
dt=0.01
num_steps=100
T=initialize_temperature_grid()
# Set boundary conditions
apply_boundary_conditions(T)
# Time-stepping loop
for step in range(num_steps):
    # Create a copy of the current temperature grid for updates
    T_new = T.copy()
    # Update the temperature at each interior grid point
    for i in range(1,nx-1):
        for j in range(1, ny-1):
            for k in range(1, nz-1):
                T_new[i,j,k] = T[i,j,k] + dt*alpha*(
                    (T[i+1,j,k]-2*T[i,j,k] + T[i-1, j, k]) / dx**2 +
                    (T[i, j+1, k] - 2*T[i, j, k] + T[i, j-1, k]) / dy**2 +
                    (T[i, j, k+1] - 2*T[i, j, k] + T[i, j, k-1]) / dz**2
                )

    # Update the grid
    T = T_new
    # Reapply boundary conditions if necessary
    apply_boundary_conditions(T)
```

This code performs the essential steps of the heat 3D algorithm by calculating and updating the temperature at each grid point for each time step.

8. Visualization

Visualizing the results helps in understanding how heat diffuses over time. Typically, this involves creating color maps or 3D plots of temperature distribution at different time points, which can be done using tools like `matplotlib` in Python.

Key Points to Remember:

- The 3D heat algorithm uses a **finite difference method** to approximate the heat equation.
- **Time stepping** allows us to simulate the progression of heat over time.
- **Boundary conditions** significantly impact the final temperature distribution.
- Choosing the right values for Δx, Δy, Δz, and Δt is crucial for accuracy and stability.

OpenMP

OpenMP (Open Multi-Processing) is a powerful, widely used library in C, C++, and Fortran that enables developers to write parallel code for shared-memory systems (like multi-core processors) without changing the core logic too much. OpenMP allows you to add "parallelism" to your program, which means splitting up tasks so multiple processors can work on them simultaneously, often leading to faster performance.

Why Use OpenMP?

When you have multiple cores (processors), running tasks in parallel can speed up computationally intensive applications, such as the **Heat Diffusion algorithm**. In this 3D Heat Diffusion project:

- **Sequential Execution** means that every point in the 3D grid is updated one by one. With a large grid, this can be slow.
- **Parallel Execution (using OpenMP)** will split the work across available CPU cores, so multiple cells in the grid are updated at once.

Key Concepts in OpenMP

1. **Directives**: OpenMP uses special commands (called directives) to specify which parts of the code should be parallelized. These directives are inserted as `#pragma` statements in C/C++.
2. **Parallel Regions**: A parallel region is a block of code that will be executed by multiple threads. You can define parallel regions using `#pragma omp parallel`.
3. **DOALL Parallelism**: In the context of loops, OpenMP's `#pragma omp for` allows us to split iterations of a loop across threads. Since each cell update in our 3D heat diffusion grid is independent of others, this is perfect for **DOALL parallelism**, where all loop iterations can be done independently in parallel.
4. **Reduction**: When a loop involves combining results (like summing up values), OpenMP provides a reduction clause to handle these cases safely across threads.
5. **Data Sharing**: OpenMP uses private and shared clauses to specify data scope. In heat diffusion, the temperature grid would generally be `shared` among threads, so all threads work on the same grid.

OpenMP and Pipelines

In a "pipelined" parallelism context, a program breaks tasks into stages, where each stage can process a different part of the data concurrently. While the 3D heat diffusion algorithm is generally a good candidate for **DOALL parallelism** (all iterations are independent and can run in parallel), pipeline parallelism is more suitable for situations where tasks need to pass intermediate results between each other in a sequence (such as image processing tasks that require sequential filters).

Implementing Heat Diffusion in OpenMP (DOALL Parallelism)

Here is a C program implementing the 3D Heat Diffusion algorithm using OpenMP with **DOALL parallelism** to process each point in the grid in parallel. Afterward, we'll measure and compare its performance against a sequential version.

Step 1: Writing the Sequential Heat Diffusion Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define NX 20
#define NY 20
#define NZ 20
#define NUM_STEPS 100
#define ALPHA 0.01

void update_temperature(double T[NX][NY][NZ]) {
    double T_new[NX][NY][NZ];
    for (int i = 1; i < NX - 1; i++) {
        for (int j = 1; j < NY - 1; j++) {
            for (int k = 1; k < NZ - 1; k++) {
                T_new[i][j][k] = T[i][j][k] + ALPHA * (
                    (T[i+1][j][k] - 2*T[i][j][k] + T[i-1][j][k]) +
                    (T[i][j+1][k] - 2*T[i][j][k] + T[i][j-1][k]) +
                    (T[i][j][k+1] - 2*T[i][j][k] + T[i][j][k-1])
                );
            }
        }
    }
    // Copy T_new back to T
    for (int i = 0; i < NX; i++)
        for (int j = 0; j < NY; j++)
            for (int k = 0; k < NZ; k++)
                T[i][j][k] = T_new[i][j][k];
}

int main() {
    double T[NX][NY][NZ] = {0};
    T[NX/2][NY/2][NZ/2] = 100.0;  // initial hot spot

    clock_t start = clock();
    for (int step = 0; step < NUM_STEPS; step++) {
        update_temperature(T);
    }
    clock_t end = clock();
    printf("Sequential Time: %f seconds\n", (double)(end - start) / CLOCKS_PER_SEC);

    return 0;
}
```

Step 2: Adding OpenMP for Parallel Execution

To enable parallelism, we'll add OpenMP `#pragma omp parallel for` to update each cell in parallel. This allows each core to update a part of the grid at the same time.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define NX 20
#define NY 20
#define NZ 20
```

```
#define NUM_STEPS 100
#define ALPHA 0.01

void update_temperature(double T[NX][NY][NZ]) {
    double T_new[NX][NY][NZ];

    #pragma omp parallel for collapse(3)
    for (int i = 1; i < NX - 1; i++) {
        for (int j = 1; j < NY - 1; j++) {
            for (int k = 1; k < NZ - 1; k++) {
                T_new[i][j][k] = T[i][j][k] + ALPHA * (
                    (T[i+1][j][k] - 2*T[i][j][k] + T[i-1][j][k]) +
                    (T[i][j+1][k] - 2*T[i][j][k] + T[i][j-1][k]) +
                    (T[i][j][k+1] - 2*T[i][j][k] + T[i][j][k-1])
                );
            }
        }
    }

    #pragma omp parallel for collapse(3)
    for (int i = 0; i < NX; i++)
        for (int j = 0; j < NY; j++)
            for (int k = 0; k < NZ; k++)
                T[i][j][k] = T_new[i][j][k];
}

int main() {
    double T[NX][NY][NZ] = {0};
    T[NX/2][NY/2][NZ/2] = 100.0;  // initial hot spot

    double start = omp_get_wtime();
    for (int step = 0; step < NUM_STEPS; step++) {
        update_temperature(T);
    }
    double end = omp_get_wtime();
    printf("Parallel Time with OpenMP: %f seconds\n", end - start);

    return 0;
}
```

Explanation of the OpenMP Code

- `#pragma omp parallel for collapse(3)`: This directive tells OpenMP to parallelize the three nested loops, effectively distributing the workload across threads for maximum efficiency. The `collapse(3)` clause allows OpenMP to handle all three loops together as a single large loop.
- `omp_get_wtime()`: OpenMP's function to get precise wall-clock time, which helps measure the parallel execution time.

DOALL Parallelism

What is DOALL?

DOALL is a type of **loop-level parallelism** where every iteration of a loop is **independent** of the others, meaning they can all be executed in parallel without needing to wait for each other. This is ideal for algorithms where operations on each element in a data set can be performed independently.

In the 3D heat diffusion algorithm:

- Each point in the grid at a given time step can be updated based on the temperatures of its neighboring points.
- The computation for each grid cell at a time step is independent of the updates to other cells because it relies only on the values from the **previous time step**.
- This independence makes the algorithm suitable for DOALL parallelism, as each grid point's temperature can be updated concurrently with others.

How DOALL Parallelism Works in the 3D Heat Algorithm

In practice, DOALL parallelism in this algorithm is achieved as follows:

1. **Loop Structure**:
   - In the code, there are nested loops iterating over the 3D grid dimensions (i.e., `i`, `j`, `k`).
   - Each iteration of the loop calculates a new temperature value for a grid point (i, j ,k) based on its neighboring points from the previous time step.
2. **Parallel Execution with OpenMP**:
   - By using OpenMP's `#pragma omp parallel for collapse(3)`, we allow each iteration (each grid cell update) to be processed by a different CPU thread.
   - **collapse(3)** merges the three nested loops into a single "loop" that OpenMP can parallelize across multiple threads, where each thread processes a subset of the grid cells.
3. **Data Sharing**:
   - The temperature grid, `T`, is declared as a **shared** array, accessible to all threads, while temporary variables within each iteration can be kept **private** to avoid race conditions.
   - OpenMP manages each thread's access to memory to prevent conflicts.

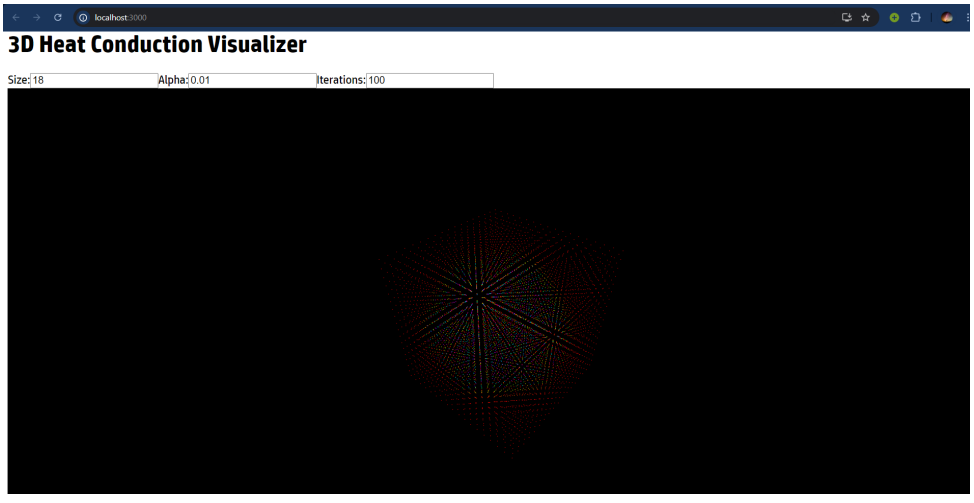
Advantages of DOALL Parallelism

- **Increased Throughput**: By allowing each CPU core to process different cells simultaneously, we reduce overall computation time significantly.
- **Scalability**: As the grid size grows, more threads can be added to handle additional cells, making DOALL highly scalable on multi-core systems.
- **Minimal Synchronization**: Since each iteration is independent, there's no need for synchronization between threads within a single time step. Only after completing a full time step, the updated values are shared.

Visualization of Heat 3D algorithm

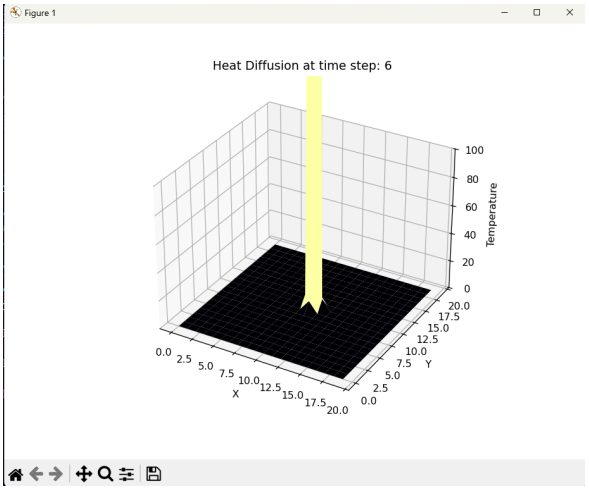
We have made a 3D simulations of how the final equilibrium state would look like. Red indicates lower temperature(0) and yellow indicates max temperature. Other colors indicates intermediate temperatures.

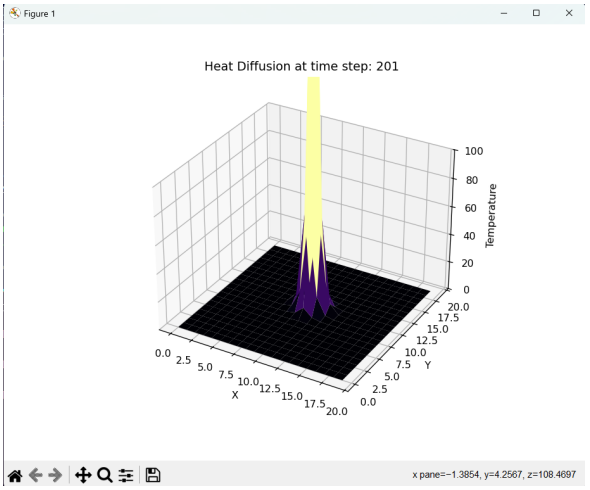
n is set to 18, alpha set to 0.01 and number of iterations are set to 100.

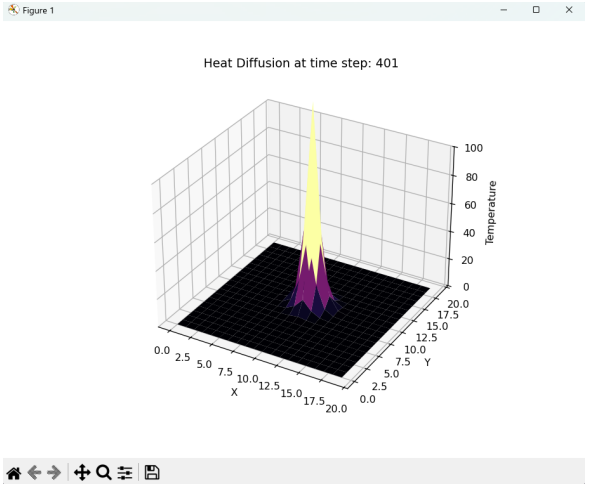


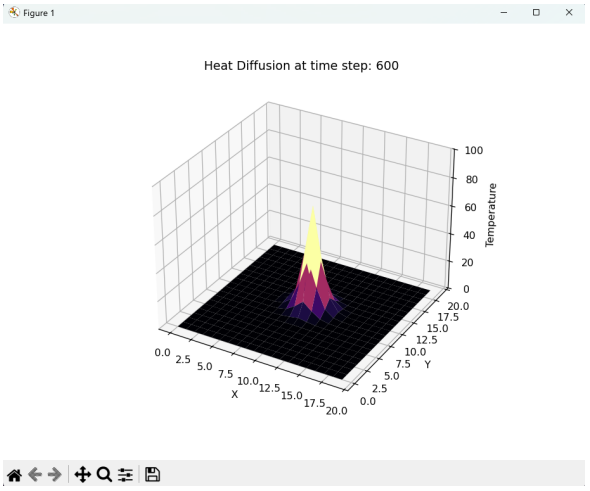
We also made a 3D graph which shows temperature flow of a cross section on x, y plane versus temperature. Note that heat conduction is happening in z axis too but we cannot visualize all the parameters together as it would require a 4D graph.

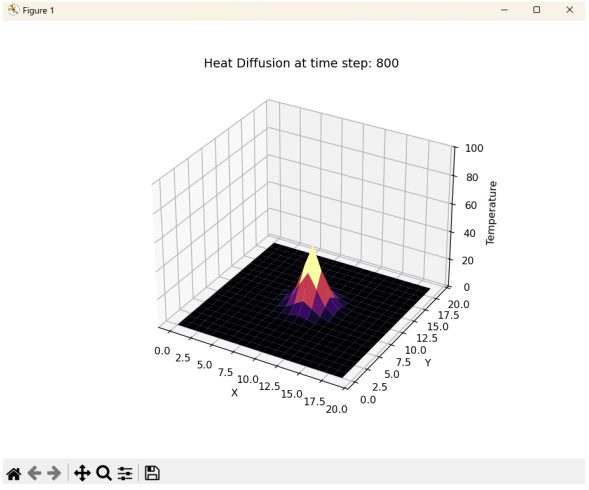
Here the grid size was 20×20×20 and initial temperature was 1000, number of iterations were 1000. In this yellow shows max temperature region, red intermediate and towards blue spectrum colors are low temperature regions. Black is zero.

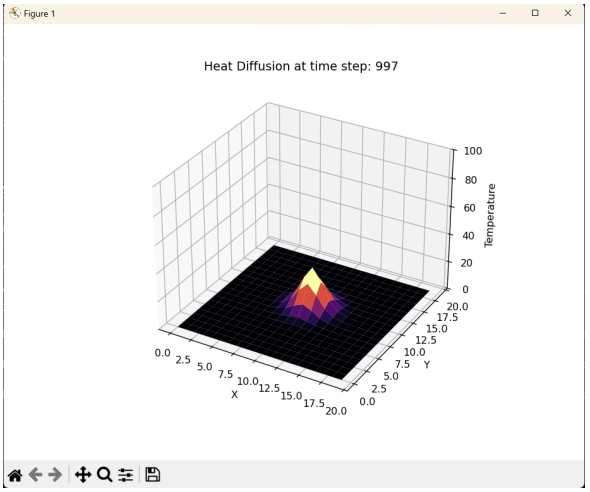


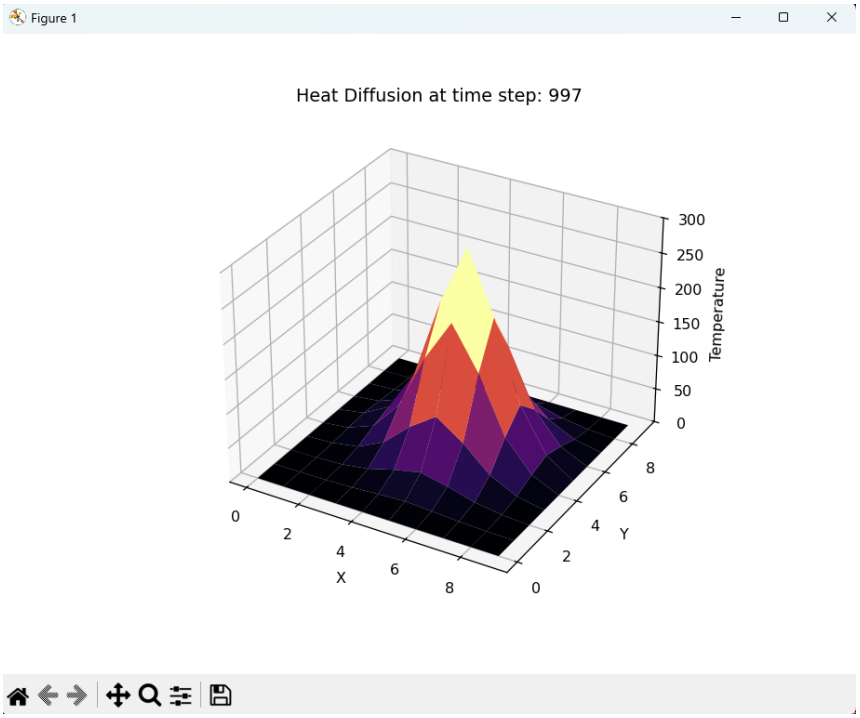












This one has 1000 iterations, grid size is 10×10×10, alpha is 0.01.

Template Code

We developed production-grade code incorporating a template, makefile, organized headers, and an efficient grid initialization process.

```
#ifndef TEMPLATE_H
#define TEMPLATE_H
#include <stdio.h>
#include <stdlib.h>
#include <numa.h>
#include <omp.h>
#define N 100  // Smaller grid for easier viewing
#define T 1000  // Number of time steps
#define ALPHA 0.01  // Diffusion coefficient
#define TILE_SIZE 4  // Tile size for cache optimization
// #define SUB_N 5  // Size of each sub-domain in i and j (assuming equal sizes for simplic
#define TEMP_SOURCE 1000.0 //Tempeeature of heat source
#endif
```

Constants Defined for all Types of Parallelism

Exploring other parallelism methods except DOALL

Tilling Parallelism

Tiling cache parallelism is a technique that is used to efficiently process a large computation by splitting it into smaller blocks (tiles) that will fit into the CPU cache. The CPU cache shows a great speedup in processing as the access time is much less than that from the main memory, so carrying out our computation there can lead to significant time savings. But putting all of the grid into the cache will cause a lot of misses because of the low cache size.

Tiling cache parallelism ensures that when each tile is being processed, the relevant and necessary data remains in the cache for as long as possible and the smaller blocks of data are chosen to achieve just that. This way we can exploit Spatial and Temporal locality very efficiently.

Again as the domain decomposition parallelism, we need to take care of boundary cells by implementing by using ghost cells which are the placeholders of the neighboring cells in other tiles.

Tiling cache parallelism can also be paired with DOALL parallelism while calculating withing a tile to further improve the performance.

```
#ifndef TILING_H
#define TILING_H
#include "template.h"
void tiled_heat3D(double*** grid, double*** new_grid) {
    for (int t = 0; t < T; t++) {
        #pragma omp parallel for collapse(3)
        for (int i_tile = 1; i_tile < N-1; i_tile += TILE_SIZE) {
            for (int j_tile = 1; j_tile < N-1; j_tile += TILE_SIZE) {
                for (int k_tile = 1; k_tile < N-1; k_tile += TILE_SIZE) {
                    // Process each tile
                    for (int i = i_tile; i < i_tile + TILE_SIZE && i < N-1; i++) {
                        for (int j = j_tile; j < j_tile + TILE_SIZE && j < N-1; j++) {
                            for (int k = k_tile; k < k_tile + TILE_SIZE && k < N-1; k++) {
                                new_grid[i][j][k] = grid[i][j][k] +
                                    ALPHA * (grid[i-1][j][k] + grid[i+1][j][k] +
                                             grid[i][j-1][k] + grid[i][j+1][k] +
                                             grid[i][j][k-1] + grid[i][j][k+1] -
                                             6 * grid[i][j][k]);
                            }
                        }
                    }
                }
            }
        }
        // Swap pointers instead of grids
        double*** temp = grid;
        grid = new_grid;
        new_grid = temp;
    }
}
#endif
```

Wavefront Parallelism

Waveform parallelism is a technique that is used in problems in which the calculation dependencies appear to move in a direction, like a wave. For such problems, in the directions perpendicular to the direction of propagation of the wave, we can visualize a wavefront or a ripple forming. Here the processes running in the wavefront are independent of each other, so it can be calculated parallelly, improving the time significantly.

In the context of heat 3d equations, we do not have dependencies of the cells with each other in the same time step. The only dependency we have is with time, that one step has to be calculated before the next one. But when we see the visualized wave propagation, we find that the wavefront or the ripple consists of only one grid instance at a time. This does not provide any potential for parallelism in context to waveform parallelism, as we would need a global synchronization of the grid for advancing every step.

But we can employ it in a different, but less efficient way. We can assume the wave to be propagating in the direction of the body diagonal of the grid at one time step. Then, we can cut diagonal planar slices of the cubic grid, and visualize that plane as the wavefront. And this will provide us ample opportunity for saving time by computing those in parallel.

```
#ifndef WAVEFRONT_H
#define WAVEFRONT_H
#include "template.h"
```

```
// Wavefront parallelism for 3D heat conduction
void wavefront_heat3D(double*** grid, double*** new_grid) {
    for (int t = 0; t < T; t++) {
        // Process diagonal "wavefronts" in increasing order
        for (int wave = 2; wave < 2 * (N - 1); wave++) {
            #pragma omp parallel for collapse(2)
            for (int i = 1; i < N - 1; i++) {
                for (int j = 1; j < N - 1; j++) {
                    int k = wave - i - j;
                    if (k >= 1 && k < N - 1) {
                        new_grid[i][j][k] = grid[i][j][k] +
                            ALPHA * (grid[i - 1][j][k] + grid[i + 1][j][k] +
                                     grid[i][j - 1][k] + grid[i][j + 1][k] +
                                     grid[i][j][k - 1] + grid[i][j][k + 1] -
                                     6 * grid[i][j][k]);
                    }
                }
            }
            // Wait for all threads to finish processing the current wavefront before movi
            #pragma omp barrier
        }
        // Swap pointers instead of grids
        double*** temp = grid;
        grid = new_grid;
        new_grid = temp;
    }
}
#endif
```

NUMA

NUMA (Non-Uniform Memory Access) Parallelism is a parallel computing technique that takes into account the memory architecture of NUMA systems. NUMA systems divide memory into nodes associated with specific CPUs. Access to memory that is “local” (on the same node as the CPU accessing it) is faster than access to “remote” memory (on a different node). Optimizing for NUMA parallelism means structuring a program so that each CPU primarily accesses memory from its local node, reducing memory access latency and improving performance.

To apply NUMA parallelism to the 3D heat equation, divide the 3D grid into subdomains that align with the NUMA memory nodes, assigning each subdomain to a specific CPU or thread group within a NUMA node. Allocate each subdomain's data in local memory to reduce cross-node traffic. Bind threads to CPUs within each node to maintain data locality. Use ghost cells along the boundaries of each subdomain to handle dependencies, periodically updating these to minimize cross-node access and improve performance.

```
#ifndef NUMA_H
#define NUMA_H
#include "template.h"
// Function to allocate 3D grid dynamically with NUMA awareness
double*** allocate_grid_numa(int node) {
    double*** grid = (double***)numa_alloc_onnode(N * sizeof(double**), node);
    for (int i = 0; i < N; i++) {
        grid[i] = (double**)numa_alloc_onnode(N * sizeof(double*), node);
        for (int j = 0; j < N; j++) {
            grid[i][j] = (double*)numa_alloc_onnode(N * sizeof(double), node);
        }
    }
    return grid;
}
```

```
// Function to free dynamically allocated NUMA grid
void free_grid_numa(double*** grid) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            numa_free(grid[i][j], N * sizeof(double));
        }
        numa_free(grid[i], N * sizeof(double*));
    }
    numa_free(grid, N * sizeof(double**));
}
// Sequential or DOALL heat conduction with NUMA-aware allocation
void numa_heat3D(double*** grid, double*** new_grid) {
    for (int t = 0; t < T; t++) {
        // #pragma omp parallel for collapse(3) // not now as sequential
        for (int i = 1; i < N-1; i++) {
            for (int j = 1; j < N-1; j++) {
                for (int k = 1; k < N-1; k++) {
                    new_grid[i][j][k] = grid[i][j][k] +
                        ALPHA * (grid[i-1][j][k] + grid[i+1][j][k] + grid[i][j-1][k] +
                                 grid[i][j+1][k] + grid[i][j][k-1] + grid[i][j][k+1] -
                                 6 * grid[i][j][k]);
                }
            }
        }
        // Swap pointers instead of grids
        double*** temp = grid;
        grid = new_grid;
        new_grid = temp;
    }
}
#endif
```

Domain Decomposition

Domain Decomposition is a parallelization technique that is employed in a case of scientific computations subjected on a large computational domain which is then divided into smaller and manageable subdomains that can be processed fairly independently. Each subdomain can be then assigned to different processors or threads and a few of the related threads can be managed if necessary, but all of them are essentially processed parallelly.

For heat3d equation we can divide the grid into several sub-grids to employ parallel processing. But we need to pay attention to some key details here. Firstly, we need to update the boundary cells of the sub-grids wherever required, so as to avoid errors. Secondly, we need to synchronize the processing of the sub-grids so that it maintains continuity across the whole implementation. We must also use ghost cells to make handling boundaries easier for the sub-grids and synchronize the data of the ghost cells as they actually represent another cell which now belongs to a different sub-grid.

```
#ifndef DOMAIN_H
#define DOMAIN_H
#include "template.h"
#include <math.h>
// Domain-decomposed parallel 3D heat conduction function
void domain_decomposed_heat3D(double*** grid, double*** new_grid) {
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int num_threads = omp_get_num_threads();
        // Calculate sub-domain bounds for each thread
```

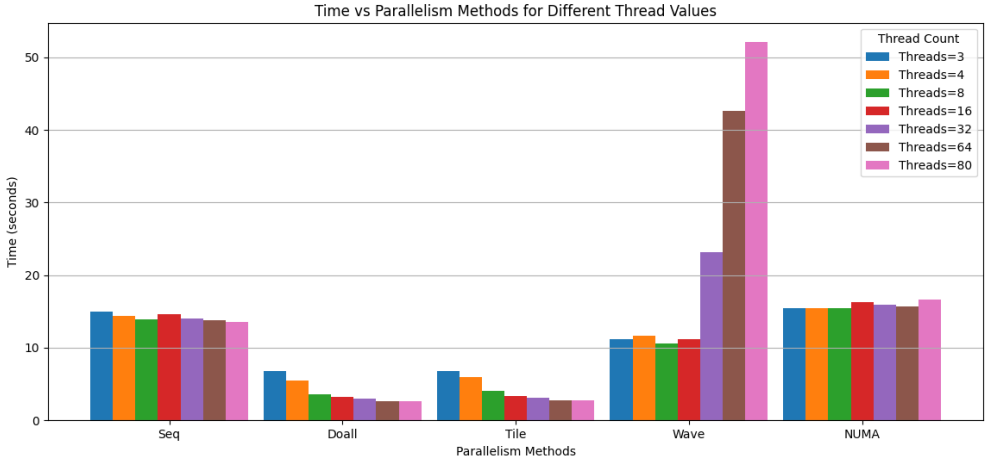
```
        int sub_domains_per_dim = (int)sqrt(num_threads); // Split threads evenly in a 2D g
        int sub_size = N / sub_domains_per_dim; // Size of each sub-domain
        int i_start = (tid / sub_domains_per_dim) * sub_size;
        int i_end = i_start + sub_size;
        int j_start = (tid % sub_domains_per_dim) * sub_size;
        int j_end = j_start + sub_size;
        // Ensure bounds do not exceed the grid dimensions
        if (i_end > N) i_end = N;
        if (j_end > N) j_end = N;
        for (int t = 0; t < T; t++) {
            // Update sub-domain assigned to each thread
            for (int i = i_start + 1; i < i_end - 1; i++) {
                for (int j = j_start + 1; j < j_end - 1; j++) {
                    for (int k = 1; k < N - 1; k++) {
                        new_grid[i][j][k] = grid[i][j][k] +
                            ALPHA * (grid[i-1][j][k] + grid[i+1][j][k] +
                                     grid[i][j-1][k] + grid[i][j+1][k] +
                                     grid[i][j][k-1] + grid[i][j][k+1] -
                                     6 * grid[i][j][k]);
                    }
                }
            }
            // Synchronize boundary data across threads
            #pragma omp barrier
            #pragma omp for collapse(2)
            for (int i = i_start; i < i_end; i++) {
                for (int j = j_start; j < j_end; j++) {
                    // Update the boundaries of each sub-domain for next iteration
                    if (i == i_start || i == i_end - 1 || j == j_start || j == j_end - 1) {
                        for (int k = 1; k < N - 1; k++) {
                            new_grid[i][j][k] = grid[i][j][k] +
                                ALPHA * (grid[i-1][j][k] + grid[i+1][j][k] +
                                         grid[i][j-1][k] + grid[i][j+1][k] +
                                         grid[i][j][k-1] + grid[i][j][k+1] -
                                         6 * grid[i][j][k]);
                        }
                    }
                }
            }
            // Swap grids after each time step
            #pragma omp single
            {
                double*** temp = grid;
                grid = new_grid;
                new_grid = temp;
            }
            #pragma omp barrier
        }
    }
}
#endif
```

Plots and Data

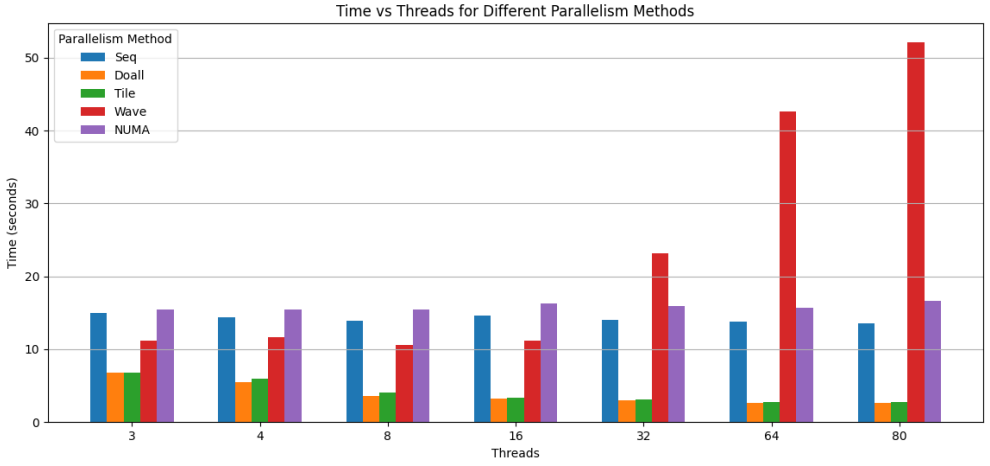
All Data after running program under various cases

Initial Parameters
N=100 (Grid Size)
T=1000 (Initial Temperature)
Alpha=0.01
Tile Size=4
Temp Source=1000.0

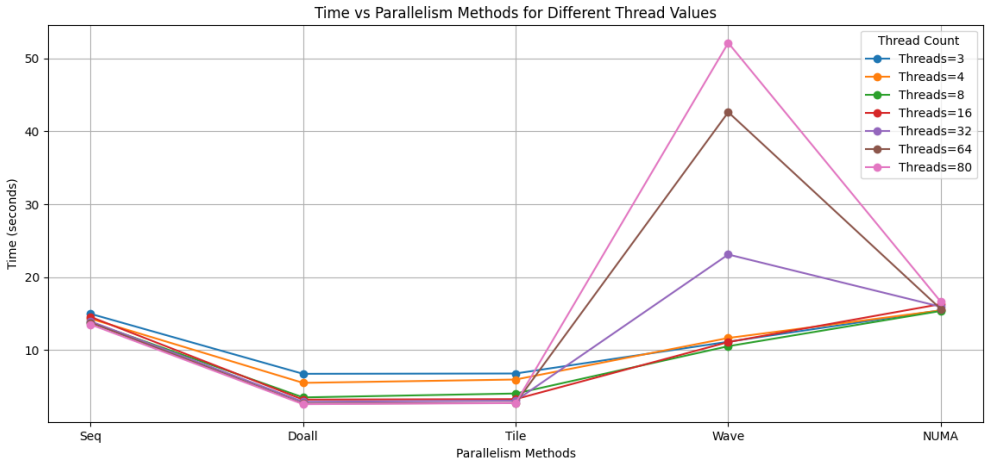
Threads	Sequential Time in sec	Do-All Time in sec	Tiled Time in sec	Wavefront Time in sec	NUMA Time in sec
3	14.9902	6.7599	6.8024	11.1542	15.4742
4	14.3205	5.5191	5.9760	11.6706	15.4536
8	13.8416	3.5255	4.0579	10.5429	15.3772
16	14.5734	3.2254	3.2865	11.1101	16.3185
32	13.9547	2.9245	3.0792	23.1167	15.9536
64	13.7679	2.6731	2.7694	42.6268	15.6342
80	13.5313	2.5979	2.7449	52.1279	16.6264



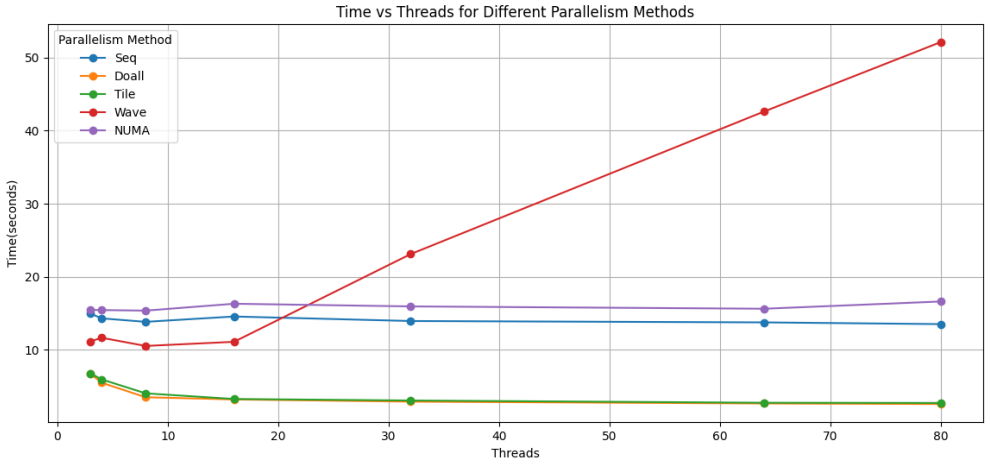
Time vs Parallelism Methods for Different Thread Values



Time vs Threads for Different Parallelism Methods



Time vs Parallelism Methods for Different Thread Values



Time vs Threads for Different Parallelism Methods

References

1. https://www.openmp.org/wp-content/uploads/openmp-examples-5.2.2-final.pdf (OpenMP and Parallelism Methods and Code Help)
2. https://www.sciencedirect.com/science/article/pii/S0955799717306604 (More about Heat3D algorithm and visualization)