HEAT SINK REPORT

Prepared By

Kharef Okba

Okba.Kharef@etu.sorbonne-universite.fr

Professor

Charles Bouillaguet

For the code provided i worked along side nicholas bell

Question 1: Why should we write the new temperatures in an intermediate array R instead of doing the computation directly in T?

Using an intermediate array R avoids overwriting the current temperature values in T before they have been used for calculations. Since the temperature of each cell depends on the temperatures of its neighboring cells, updating T in-place would introduce dependencies that could result in incorrect calculations. By writing the new temperatures to R, we ensure that all calculations for a given timestep are based on the previous state stored in T.

Question 2: In the algorithm, what can be parallelized?

Several parts of the algorithm can be parallelized:

1. Temperature Updates: The computation of new temperatures for each cell in the 3D grid can be parallelized because each cell's update is independent of others within the same timestep.
2. Max and Error Calculations: The computation of Tmax and ε involves reductions, which can also be parallelized using collective operations.
3. Data Exchange: The exchange of "halo" or boundary values between neighboring subdomains can be done in parallel by overlapping communication with computation.

Question 3: What type of load balancing would you favor, and why?

A static load balancing approach is preferred in this context because the workload for each cell is predictable and uniform. Dividing the grid equally among processors ensures that each processor has approximately the same number of cells to process, minimizing idle time. Dynamic load balancing is unnecessary unless there are non-uniform workloads or unpredictable computational demands.

Question 4: How would you partition the data on the different processors in this context?

A common approach is to partition the 3D grid along one or more dimensions:

1. 1D Partitioning: Divide the grid along one axis (e.g., the z-axis). Each processor gets a contiguous block of xy-planes. This is simple to implement but may not scale well for very large grids or high processor counts.
2. 2D Partitioning: Divide the grid into subdomains along two axes (e.g., x and z). Each processor handles a rectangular prism of cells. This provides better load balancing and reduces the size of the "halo" regions compared to 1D partitioning.

For this problem, 1D partitioning along the z-axis is sufficient since the number of processors is assumed to divide o (the number of slices along z). Each processor handles an equal number of xy-planes.

Question 5: What collective communication routines would you use to handle the computation Tmax and ε? And for eventually saving T?

1. For Tmax and ε:
    - Use MPI_Reduce to compute the global maximum temperature (Tmax) and the global error (ε).
    - Alternatively, MPI_Allreduce can be used if all processors need the results.
2. For Saving T:
    - Use MPI_Gather to collect all subdomains onto a single processor for writing to a file.
    - For large grids, MPI_File_write_all can be used to write data directly to a file in parallel, leveraging MPI-IO for efficiency.

Question 6: Implement a parallel algorithm with blocking sending of messages, using MPI_Sendrecv for exchanging "halos."

To exchange halos, each processor communicates with its neighbors along the z-axis (assuming 1D partitioning). The MPI_Sendrecv function is used for both sending and receiving boundary data:

```
void exchange_halos(double *T, int n, int m, int local_o, int rank, int size, MPI_Comm comm) {
    MPI_Status status;
    int above = rank - 1;
    int below = rank + 1;

    // Send to below, receive from above
```

```
    if (below < size) {
        MPI_Sendrecv(&T[(local_o - 1) * n * m], n * m, MPI_DOUBLE,
below, 0,
                     T, n * m, MPI_DOUBLE, above, 0, comm, &status);
    }

    // Send to above, receive from below
    if (above >= 0) {
        MPI_Sendrecv(&T[0], n * m, MPI_DOUBLE, above, 1,
                     &T[local_o * n * m], n * m, MPI_DOUBLE, below, 1,
comm, &status);
    }
}
```

Question 7 (Bonus): Implement the algorithm with non-blocking primitives, and in such a way that computing time and communication time overlap. Analyze the performances of this algorithm.

Using non-blocking communication (e.g., MPI_Isend and MPI_Irecv), we can overlap communication with computation. The boundary cells are sent/received while the interior cells are being updated:

```
void exchange_halos_nonblocking(double *T, int n, int m, int local_o,
int rank, int size, MPI_Comm comm) {
    MPI_Request requests[4];
    int above = rank - 1;
    int below = rank + 1;

    // Non-blocking send/receive to/from below
    if (below < size) {
```

```
        MPI_Isend(&T[(local_o - 1) * n * m], n * m, MPI_DOUBLE,
below, 0, comm, &requests[0]);
        MPI_Irecv(T, n * m, MPI_DOUBLE, above, 0, comm,
&requests[1]);
    }

    // Non-blocking send/receive to/from above
    if (above >= 0) {
        MPI_Isend(&T[0], n * m, MPI_DOUBLE, above, 1, comm,
&requests[2]);
        MPI_Irecv(&T[local_o * n * m], n * m, MPI_DOUBLE, below, 1,
comm, &requests[3]);
    }

    // Perform interior updates while waiting for halos
    update_interior_cells(T, n, m, local_o);

    // Wait for communication to complete
    MPI_Waitall(4, requests, MPI_STATUSES_IGNORE);
}
```

Performance Analysis: Overlapping communication with computation reduces the idle time caused by communication delays. This is particularly beneficial for large grids and systems with high communication latency.

Question 8 (Extra Bonus): Implement another way of partitioning the data, and compare with the previous results.

Another way to partition the data is 2D partitioning along the x and z axes. Each processor handles a block of the grid, reducing the number of cells in the "halo" regions.

Advantages:

- Reduces the total communication volume by minimizing the size of boundary regions.
- Better load balancing for large processor counts.

Comparison:

- 1D Partitioning: Simpler to implement, lower communication overhead for small processor counts.
- 2D Partitioning: More efficient for large processor counts, but introduces additional complexity in communication routines (e.g., diagonal neighbors).