TDT4200 Parallel programming

PS6

Maren Wessel-Berg & Claudi Lleyda Moltó

October 2023

Practical information

Published: 24/10/23

Deadline: 7/11/23 at 22:00

Evaluation: Graded (10%)

- Completing the problem set is **mandatory** and it will count towards 10% of your final course grade.
- The work must be done **individually** and without help from anyone but the TDT4200 staff.
- **Reference** all sources found on the internet or elsewhere.
- The **requirements**, and **how and what to deliver** is explained in the problem set description found on BlackBoard.
- Start the exercises early!

Where can you get help with the assignment?

- **Recitation lecture**: introduction to the problem set

  (Today)
  Slides will be made available online.
- **TA hours**: ask questions in person

  Friday, October 27th, 10:00–12:00 in Cybele
  Monday, October 31st, 13:00–15:00 in Cybele
  Friday, November 3rd, 10:00–12:00 in Cybele
  Monday, November 6th, 13:00–15:00 in Cybele
- **Piazza**: question forum

  Ask questions any time (but give us time to answer).
  Select the ps6 folder for questions related to this problem set.
  Do not post full or partial solutions!

Implicit method to solve the heat equation

Consider the 2D heat equation

$$\frac{\partial u}{\partial t} = K\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right).$$

We can discretize it as

$$\begin{aligned}\frac{u_{i,j}^k - u_{i,j}^{k-1}}{\Delta t} &= K\left(\frac{u_{i-1,j}^k - 2u_{i,j}^k + u_{i+1,j}^k}{\Delta x^2} + \frac{u_{i,j-1}^k - 2u_{i,j}^k + u_{i,j+1}^k}{\Delta y^2}\right) \\ &= K\left(\frac{u_{i-1,j}^k + u_{i+1,j}^k + u_{i,j-1}^k + u_{i,j+1}^k - 4u_{i,j}^k}{\Delta x^2}\right),\end{aligned}$$

where we have assumed $\Delta x = \Delta y$. By isolating $u_{i,j}^{k-1}$ we get

$$u_{i,j}^{k-1} = \left(1+4K\frac{\Delta t}{\Delta x^2}\right)u_{i,j}^k - K\frac{\Delta t}{\Delta x^2}\left(u_{i-1,j}^k + u_{i+1,j}^k + u_{i,j-1}^k + u_{i,j+1}^k\right).$$

Implicit method to solve the heat equation

We can clean up the expression

$$u_{i,j}^{k-1} = \left(1+4K\frac{\Delta t}{\Delta x^2}\right)u_{i,j}^{k} - K\frac{\Delta t}{\Delta x^2}\left(u_{i-1,j}^{k}+u_{i+1,j}^{k}+u_{i,j-1}^{k}+u_{i,j+1}^{k}\right)$$

into

$$u_{i,j}^{k-1} = Du_{i,j}^{k} + A\left(u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j+1}^{k}\right)$$

by setting

$$D = 1 + 4K\frac{\Delta t}{\Delta x^2}, \qquad A = -K\frac{\Delta t}{\Delta x^2}.$$

From here we get that the equality we are looking for is

$$u_{i,j}^{k} = \frac{u_{i,j}^{k-1} - A\left(u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j+1}^{k}\right)}{D}.$$

This is what we use, in conjunction with the Red/Black Gauss-Seidel method, to solve our equation!

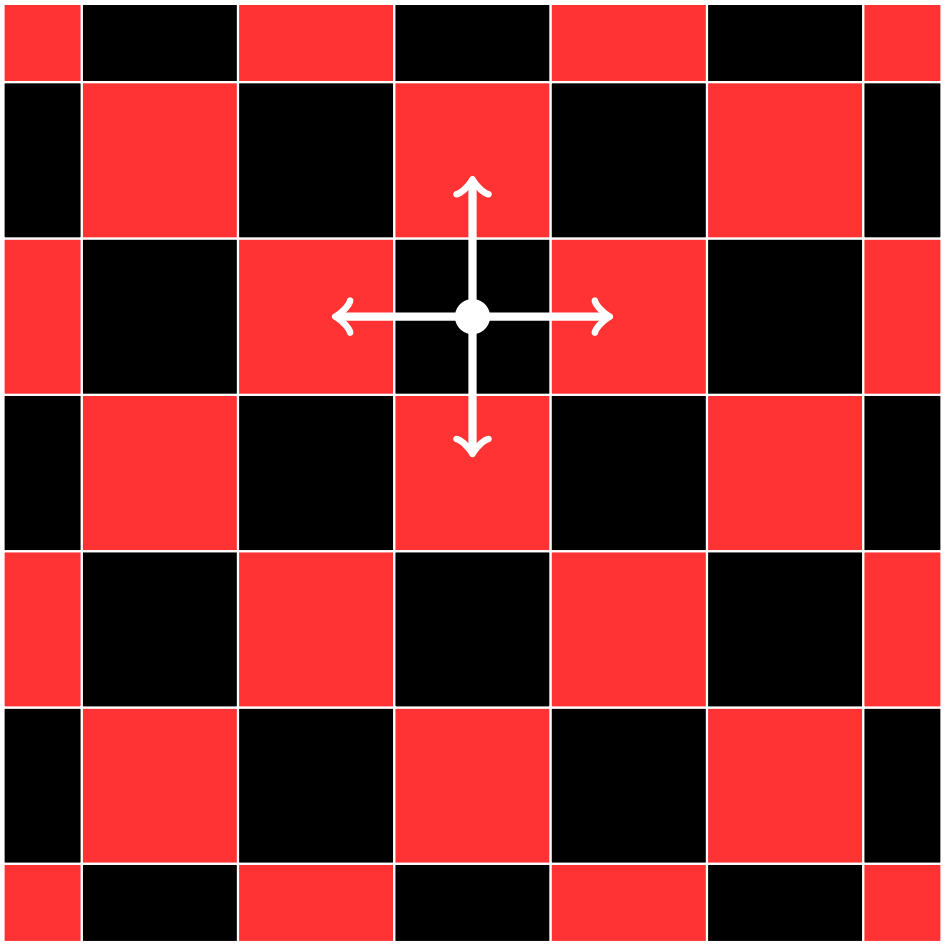
Red/Black Gauss-Seidel algorithm

Intuitive notion

- For this exercise we will revisit the Red/Black Gauss-Seidel implicit solver from PS1.
- The Gauss-Seidel algorithm improves the convergence speed of the Jacobi algorithm by updating the calculated values as we iterate.
- Unfortunately, this introduces a data dependency, heavily hindering parallelization potential.
- The Red/Black Gauss-Seidel algorithm dampens the data dependency by performing the calculations in an specific order.

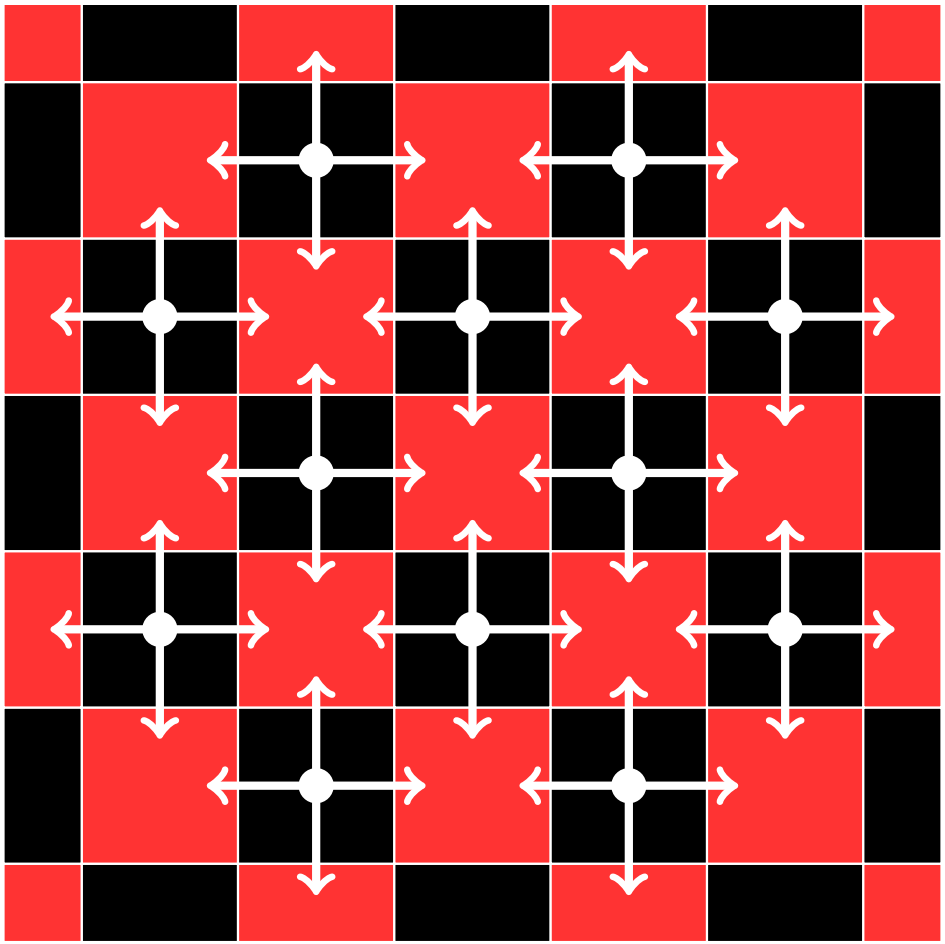
Red/Black Gauss-Seidel algorithm

To calculate the values for the selected node we only need data from its four direct neighbours (and the node itself).



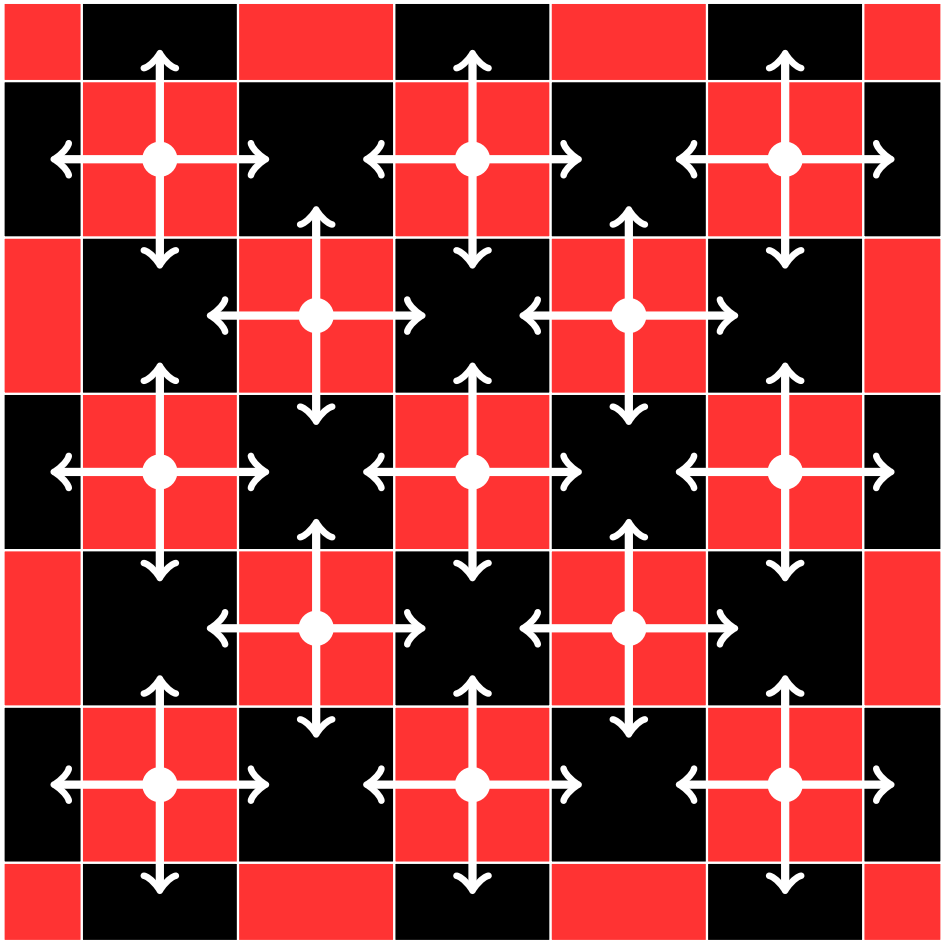
Red/Black Gauss-Seidel algorithm

We can calculate the values for all the black nodes independently from each other, without any data races.



Red/Black Gauss-Seidel algorithm

The same is true for the red nodes!



Gauss-Seidel algorithm example

Consider the linear system

$$\begin{bmatrix} 3 & 1 & 0 \\ 2 & 8 & 1 \\ -1 & -2 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \\ 3 \end{bmatrix}$$

From which we get

$$\begin{aligned} x_{k+1} &= (2 - y_k)/3 \\ y_{k+1} &= (6 - 2x_{k+1} - z_k)/8 \\ z_{k+1} &= (3 + x_{k+1} + 2y_{k+1})/4. \end{aligned}$$

If we chose our first approximation to be $(0, 0, 0)$ we get

$$\begin{aligned} x_1 &= (2 - 0 - 0)/3 = 0.6667 \\ y_1 &= (6 - 2 * 0.6667 - 0)/8 = 0.5833 \\ z_1 &= (3 + 0.6667 + 2 * 0.5833)/4 = 1.2083. \end{aligned}$$

Gauss-Seidel algorithm example

We can continue

$$x_2 = (2 - 0.5833)/3 = 0.4722$$
$$y_2 = (6 - 2 * 0.4722 - 1.2083)/8 = 0.4809$$
$$z_2 = (3 + 0.4722 + 2 * 0.4809)/4 = 1.1085,$$

from which we get

$$x_3 = (2 - 0.4809)/3 = 0.5064$$
$$y_3 = (6 - 2 * 0.5064 - 1.1085)/8 = 0.4848$$
$$z_3 = (3 + 0.5064 + 2 * 0.4848)/4 = 1.1190,$$

and lastly

$$x_4 = (2 - 0.4848)/3 = 0.5051$$
$$y_4 = (6 - 2 * 0.5051 - 1.1190)/8 = 0.4839$$
$$z_4 = (3 + 0.5051 + 2 * 0.4839)/4 = 1.1182,$$

where we begin to reach convergence.

Cooperative groups

Synchronization

- We can perform the calculations pertaining to squares of a certain color in parallel in the GPU, but we must ensure that we wait for all the calculations under that color to be complete before starting the next color.
- This means we must synchronize our tasks in between.
- Typically, we can only control one block at a time, but we will require many blocks to execute at every time step.
- We could artificially synchronize our calculations by launching two different kernels, one after the other, but that would be inefficient.

```
compute_red_kernel<<<grid, blocks>>>(domain);
// Artifical sync
compute_black_kernel<<<grid, blocks>>>(domain);
```

Cooperative groups

Grid synchronization

- Fortunately, we can synchronize an entire grid of blocks at once using *cooperative groups*.

```
namespace cg = cooperative_groups;

void __global__ kernel(real_t *domain)
{
    cg::grid_group grid = cg::this_grid();
    compute_red(domain);
    grid.sync();
    compute_black(domain);
}
```

- No such thing as a free lunch :(It has some caveats.

Cooperative groups

Grid synchronization considerations

This approach requires all blocks to be present in the GPU **at the same time**.

- Meaning, we cannot launch a grid with more blocks than our GPU can physically fit.
- You will have to look at the specs of your GPU to figure out this limit (or query-it during runtime).
- For this problem set, we have set the default domain size to be small enough to fit in the GPUs available to you in the cluster.

Cooperative groups

Grid synchronization considerations

The `grid.sync()` function must be reached by **all** threads in the grid, even those you may have wished to discard.

- No more early returns.

```
void __global__ kernel()
{
    cg::grid_group grid = cg::this_grid();
    if (is_out_of_bounds())
        return;

    do_stuff();
    grid.sync(); // <- This will fail!
    other_stuff();
}
```

Cooperative groups

Grid synchronization considerations

The kernel launch instruction will require different, more verbose, syntax. You must use

```
void *args[] = {
    (void*) &arg1,
    (void*) &arg2,
    (void*) &arg3,
    (void*) &arg4,
};
cudaLaunchCooperativeKernel((void *) kernel,
                            grid, blocks, args);
```

instead of

```
kernel<<<grid, blocks>>>(arg1, arg2, arg3, arg4);
```

which will crash when trying to run `grid.sync()`.

Your tasks

Implementation details

Most tasks in this problem are fairly similar to the ones found in PS5.

- Keep in mind that for this approach you do not need a duplicate of the simulation domain in memory, as everything is done in-place.
- This is also reflected in the sequential implementation, so you can use that as a reference.

Your tasks

Implementation details

The `time_step` kernel will have to be rewritten, both to accommodate the grid synchronization and the new algorithm.

- Be careful with early returns for threads with indices outside the simulation domain! You will have to find an alternative to this.
- Remember that all threads in the grid must reach the synchronization instruction.

The kernel launch instruction will also need to change with respect to the last problem set, to accommodate for the cooperative threads syntax.

- You will need to think about how to send arguments such as `N`, `M` and `dt` to the GPU with this syntax.