Projektarbeit PC1

Dokumentation

Maurice Wenig

Inhaltsverzeichnis

1 Vorgehen

Das initiale Feld wird gleichmäßig auf alle Prozesse aufgeteilt. In jedem Prozess werden die neuen Temperaturen im lokalen *chunk* berechnet. Dafür gibt es einen Austausch der aktuellen Werte mit den Nachbarprozessen. Diese Werte werden in den Ghost Cells in einem Halo der Breite g um den lokalen *chunk* gespeichert. Ab hier werden die Ghost Cells als Teil des lokalen Chunks angesehen. Der Teil des Chunks, der keine Ghost Cells beinhaltet, wird als innerer Chunk bezeichnet. Am Ende der Berechnung werden die innneren Chunks der einzelnen Prozesse wieder zusammengesetzt.

```
1: split_up_domain()
2: for i = 0 → n_iterations do
3:     if i%g == 0 then
4:         exchange_ghost_cells()
5:     calculate()
6: collect()
```

1.1 Kommunikation

Die angewandten Techniken werden größtenteils dem Paper "Ghost Cell Pattern" von Kjolstad and Snir entnommen. Es wird ein Deep Halo benutzt und die Corner Cells sollen auch effizient übertragen werden. Da die Ost-West-Kommunikation immer vor der Nord-Süd-Kommunikation passieren muss, damit die Ecken richtig übertragen werden, wird zwischen der Kommunikation von den beiden Richtungen gewartet.

```
1: irecv_east()
2: irecv_west()
3: isend_east()
4: isend_west()
5: wait()
6: irecv_north()
7: irecv_south()
8: isend_north()
9: isend_south()
10: wait()
```

Falls ein Nachbar in eine Richtung nicht existiert, werden die Ghost Cells mit Padding gefüllt. Dabei wird immer der nächste Wert des inneren Chunks kopiert.

1.2 Berechnung

Es werden immer nur so viele Zellen berechnet, wie für den nächsten Schritt benötigt werden. Dafür wird eine *border* eingeführt. Am Anfang umfasst die *border* den ganzen Chunk, inklusive Halo, bis auf den äußersten Ring. Am Ende umfasst die *border* nur noch den inneren Chunk.

Um die neuen Werte zu berechnen werden zwei Arrays benutzt. Eines, das die alten Werte enthält

und eines, das die Ergebnisse der Berechnung enthält. Als Vorbereitung für den nächsten Schritt werden am Ende der Berechnung die Arrays vertauscht.

1: *border* := adapt_border()
2: **for** x, y in the *border* **do**
3: results$[x, y]$:= calculation_step(x, y) ▷ calculation_step uses old_values
4: swap(results, old_values)

2 Implementierung

2.1 Chunks

Zu dem inneren Chunk kommt noch das Ghost-Block-Halo dazu:

```
        // local chunk with ghost blocks
        double* l_chunk = (double*) malloc((chunk_dimensions[X_AXIS] + 2 * g) * (chunk_dimensions[Y_AXIS] + 2 * g) *
↪  sizeof(double));
        // buffer for local chunk
        double* l_chunk_buf = (double*) malloc((chunk_dimensions[X_AXIS] + 2 * g) * (chunk_dimensions[Y_AXIS] + 2 *
↪  g) * sizeof(double));
        // fill the local chunk
        fill_local_chunk(rank, n_processes, l_chunk, chunk_dimensions, u1, size, g);
```

2.2 Kommunikation

2.2.1 Ghost Cells

Für bessere Lesbarkeit werden die Axen `X_AXIS`, `Y_AXIS` und die Richtungen `EAST`, `WEST`, `NORTH`, `SOUTH` definiert. Den Richtungen werden relative Positionen im Gitter zugewiesen. Außerdem werden ein `MAIN_RANK` definiert, der alles sammelt und ausgibt, sowie ein undefinierter Rang `UNDEFINED_RANK`, der für Nachbarn außerhalb des Gitters steht.

```
// our problem is 2-dimensional
#define N_DIMENSIONS 2
#define X_AXIS 0
#define Y_AXIS 1

// neighbours
#define N_NEIGHBOURS 4
#define EAST 0
#define WEST 1
#define NORTH 2
#define SOUTH 3

// which directions are the directions actually?
int diff_directions[N_NEIGHBOURS][N_DIMENSIONS] = {
    // EAST (X,Y)
    { 1, 0 },
    // WEST (X,Y)
    { -1, 0 },
    // NORTH (X,Y)
    { 0, -1 },
    // SOUTH (X,Y)
    { 0, 1 },
};

// rank for printing and stuff
#define MAIN_RANK 0
#define UNDEFINED_RANK -1
```

Um die Blöcke von Ghost Cells effizient zu verschicken, werden Vektoren von MPI verwendet.

```
        // make the types for communication
        int block_stride, block_count, block_length;
        // exchange all known data across the vertical line, g wide. (stride is the whole horizontal dimension)
        MPI_Datatype vertical_border_t;
        get_vector_properties(EAST, chunk_dimensions, g, &block_count, &block_length, &block_stride);
        MPI_Type_vector(block_count, block_length, block_stride, MPI_DOUBLE, &vertical_border_t);
```

```
        MPI_Type_commit(&vertical_border_t);
        // exchange all data across the horizontal line, whole horizontal line wide, g blocks. (stride is the whole
↪  horizontal dimension)
        MPI_Datatype horizontal_border_t;
        get_vector_properties(NORTH, chunk_dimensions, g, &block_count, &block_length, &block_stride);
        MPI_Type_vector(block_count, block_length, block_stride, MPI_DOUBLE, &horizontal_border_t);
        MPI_Type_commit(&horizontal_border_t);

void get_vector_properties(int direction, int* chunk_dimensions, int g, int* block_count, int* block_length, int*
↪  block_stride) {
    if (direction == EAST || direction == WEST) {
        *block_count = chunk_dimensions[Y_AXIS];
        *block_length = g;
        *block_stride = chunk_dimensions[X_AXIS] + 2 * g;
        return;
    }
    if (direction == NORTH || direction == SOUTH) {
        *block_count = g;
        *block_length = chunk_dimensions[X_AXIS] + 2 * g;
        *block_stride = chunk_dimensions[X_AXIS] + 2 * g;
        return;
    }
}
```

Die Punkte, an denen Blöcke anfangen, werden hier definiert:

```
// start indices for ghost block buffers
void set_comm_indices(int* send_buffer_start, int* recv_buffer_start, int* chunk_dimensions, int g) {
    send_buffer_start[EAST] = chunk_index(chunk_dimensions[X_AXIS], g);
    send_buffer_start[WEST] = chunk_index(g, g);
    send_buffer_start[NORTH] = chunk_index(0, g);
    send_buffer_start[SOUTH] = chunk_index(0, chunk_dimensions[Y_AXIS]);

    recv_buffer_start[EAST] = chunk_index(chunk_dimensions[X_AXIS] + g, g);
    recv_buffer_start[WEST] = chunk_index(0, g);
    recv_buffer_start[NORTH] = chunk_index(0, 0);
    recv_buffer_start[SOUTH] = chunk_index(0, chunk_dimensions[Y_AXIS] + g);
}
```

Mit `neighbours[direction]` als Rang des Nachbars in der jeweiligen Richtung wird dann wie folgt kommuniziert:

```
void send_ghosts(int direction, int* neighbours, int* send_buffer_start, double* grid, MPI_Datatype border_type, int*
↪  array_of_requests, int* current_request, int tag) {
    // do nothing if the neighbour does not exist
    if (neighbours[direction] == UNDEFINED_RANK) return;
    // send from non-ghost-zone-end
    MPI_Isend(&grid[send_buffer_start[direction]], 1, border_type, neighbours[direction], tag,
        MPI_COMM_WORLD, &array_of_requests[*current_request]);
    // update current index because communication was succesful
    ++(*current_request);
}

void recv_ghosts(int direction, int* neighbours, int* recv_buffer_start, double* grid, MPI_Datatype border_type, int*
↪  array_of_requests, int* current_request, int* chunk_dimensions, int g) {
    // pad if the neighbour does not exist
    if (neighbours[direction] == UNDEFINED_RANK) {
        pad(direction, recv_buffer_start, grid, chunk_dimensions, g);
        return;
    }
    // receive in ghost-zone
    MPI_Irecv(&grid[recv_buffer_start[direction]], 1, border_type, neighbours[direction], MPI_ANY_TAG,
        MPI_COMM_WORLD, &array_of_requests[*current_request]);
    // update current index because communication was succesful
```

```
    ++(*current_request);
}
```

Padding:

```
void pad(int direction, int* recv_buffer_start, double* grid, int* chunk_dimensions, int g) {
    // basically build the properties of the vector types again
    int block_stride, block_count, block_length;
    get_vector_properties(direction, chunk_dimensions, g, &block_count, &block_length, &block_stride);
    // find out where the ghost blocks start
    int chunk_start_x = recv_buffer_start[direction] % (chunk_dimensions[X_AXIS] + 2 * g);
    int chunk_start_y = recv_buffer_start[direction] / (chunk_dimensions[X_AXIS] + 2 * g);
    // current coordinates in the chunk
    int chunk_x, chunk_y;
    // pixel that we take our vale from (reference)
    int ref_x, ref_y;
    for (int block_y = 0; block_y < block_count; ++block_y) {
        for (int block_x = 0; block_x < block_length; ++block_x) {
            chunk_x = chunk_start_x + block_x;
            chunk_y = chunk_start_y + block_y;
            ref_x = chunk_x;
            ref_y = chunk_y;
            // trim to inner field -> ref is going to be the nearest pixel in the field
            if (ref_x > chunk_dimensions[X_AXIS] + g - 1) ref_x = chunk_dimensions[X_AXIS] + g - 1;
            if (ref_x < g) ref_x = g;
            if (ref_y > chunk_dimensions[Y_AXIS] + g - 1) ref_y = chunk_dimensions[Y_AXIS] + g - 1;
            if (ref_y < g) ref_y = g;
            // now pad the ghost block with the nearest value in the field
            grid[chunk_index(chunk_x, chunk_y)] = grid[chunk_index(ref_x, ref_y)];
        }
    }
}
```

2.2.2 Collect

Auch hier werden wieder Vektoren verwendet:

```
        // type for all the inner values (non-ghost-blocks)
        MPI_Datatype chunk_inner_values_t;
        MPI_Type_vector(chunk_dimensions[Y_AXIS], chunk_dimensions[X_AXIS], chunk_dimensions[X_AXIS] + 2 * g,
↪   MPI_DOUBLE, &chunk_inner_values_t);
        MPI_Type_commit(&chunk_inner_values_t);
```

Die inneren Chunks werden dann hintereinander in einen Buffer geschrieben. Die Werte im Buffer werden für das Ergebnis umgeordnet.

```
void collect(double* global_grid, int size, double* local_chunk, int* chunk_dimensions, int* n_processes, int g,
↪   MPI_Datatype chunk_inner_values_t) {
    // first collect all the data from the chunks
    double* recv_buf = (double*) malloc(size * size * sizeof(double));
    MPI_Gather(&local_chunk[chunk_index(g, g)], 1, chunk_inner_values_t, recv_buf, chunk_dimensions[X_AXIS] *
↪   chunk_dimensions[Y_AXIS], MPI_DOUBLE, MAIN_RANK, MPI_COMM_WORLD);
    // then arrange them correctly
    int recv_buf_index, global_grid_index;
    // number of all the processes
    int total_ranks = n_processes[X_AXIS] * n_processes[Y_AXIS];
    // coords of the current rank
    int coords[N_DIMENSIONS];
    // offset in the global grid of the chunk that the current rank is processing
    int offset[N_DIMENSIONS];
    for (int rank = 0; rank < total_ranks; ++rank) {
        for (int chunk_y = 0; chunk_y < chunk_dimensions[Y_AXIS]; ++chunk_y) {
            for (int chunk_x = 0; chunk_x < chunk_dimensions[X_AXIS]; ++chunk_x) {
```

```
                // recv buf if now basically a 3-dimensional array with dimensions (rank, y, x)
                recv_buf_index = chunk_dimensions[X_AXIS] * (chunk_dimensions[Y_AXIS] * rank + chunk_y) + chunk_x;
                // index in the global array is defined by the coordinates that the chunk is written to
                get_coords(rank, n_processes, coords);
                // offset of the chunk in the global array
                offset[X_AXIS] = coords[X_AXIS] * chunk_dimensions[X_AXIS];
                offset[Y_AXIS] = coords[Y_AXIS] * chunk_dimensions[Y_AXIS];
                global_grid_index = global_index(offset[X_AXIS] + chunk_x, offset[Y_AXIS] + chunk_y);
                // now actually write it
                global_grid[global_grid_index] = recv_buf[recv_buf_index];
            }
        }
    }
}
```

2.3 Berechnung

Berechnung nach Aufgabenstellung mit `FACTOR` $= \alpha \cdot \frac{\Delta t}{h^2}$.

```
                // narrow the border
                border = i % g;
                // update the grid
                for (int y = 1 + border; y < chunk_dimensions[Y_AXIS] + 2 * g - (1 + border); ++y) {
                        for (int x = 1 + border; x < chunk_dimensions[X_AXIS] + 2 * g - (1 + border); ++x) {
                                l_chunk_buf[chunk_index(x, y)] = l_chunk[chunk_index(x, y)] + FACTOR *
↪  (l_chunk[chunk_index(x + 1, y)] + l_chunk[chunk_index(x, y + 1)] + l_chunk[chunk_index(x - 1, y)] +
↪  l_chunk[chunk_index(x, y - 1)] - 4 * l_chunk[chunk_index(x, y)]);
                        }
                }
                swap(&l_chunk, &l_chunk_buf);
```

3 Benutzung

Argumente, die übergeben werden müssen:

- Größe des zu berechnenden Feldes
- Anzahl der Iterationen
- Anzahl der Prozesse in X-Richtung
- Anzahl der Prozesse in Y-Richtung

Parameter, die angepasst werden können:

- Ausgabedatei
- Breite des Ghost Cell Halos
- Abstand der Iterationen, in denen ein Zwischenstand gespeichert wird

Literatur

Fredrik Berg Kjolstad and Marc Snir. Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns - ParaPLoP '10*. ACM Press, 2010. doi: 10.1145/1953611.1953615.