Laplacian relaxation optimization

DIETZ T., JABER A., DONNENFELD T.

04 November 2025

1 Algorithmic Changes

1.1 Matrix Update

The original implementation copied the new array into the old one element by element:

```
void laplace_copy(double *A, double *Anew, int n, int m) {
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < m; i++) {
            A[j*m + i] = Anew[j*m + i];
        }
    }
}
```

Listing 1: initial_code.c – full matrix copy

A better version is to simply swap the pointers to avoid the whole copy process. We just have to be careful with the borders of the uninitialized matrix, which by default have not been set to their constant values.

```
double *Atmp = A;
A = Anew;
Anew = Atmp;
```

Listing 2: final_code.c – pointer swap

This change prevents an entire pass over the matrix for each iteration.

1.2 Error Computation

In the initial implementation, the error was computed in a separate function, requiring a full matrix reading after each iteration.

```
double laplace_error(double *A, double *Anew, int n, int m) {
    double error = 0.0;
    for (int j = 1; j < n - 1; j++) {
        for (int i = 1; i < m - 1; i++) {
            error = fmax(error, fabs(Anew[j*m + i] - A[j*m + i]));
        }
    }
    return error;
}
```

Listing 3: initial_code.c – separate error computation

In the optimized version, the error is computed directly during the relaxation step, removing the need for an extra matrix pass:

```
#pragma omp parallel for reduction(max:error) collapse(2)
for (int j = 1; j < n - 1; j++) {
    for (int i = 1; i < m - 1; i++) {
        Anew[j*m + i] = 0.25f * (A[j*m + i + 1] + A[j*m + i - 1] +
                                 A[(j-1)*m + i] + A[(j+1)*m + i]);
        error = fmaxf(error, fabsf(Anew[j*m + i] - A[j*m + i]));
    }
}
```

Listing 4: final_code.c – integrated error computation

2 Code-Level Optimizations

2.1 Matrix index access swapping

Changing the way data from the matrices is accessed so that less memory fetches are made. This is a simple for loop swap.

```
for ( i=1; i < m-1; i++ )
  for ( j=1; j < n-1; j++ )
    out[j*m+i]= stencil(in[j*m+i+1], in[j*m+i-1], in[(j-1)*m+i], in[(j+1)*m+i]);
```

Listing 5: initial_code.c

```
for ( j=1; j < n-1; j++ ) {
  for ( i=1; i < m-1; i++ ) {  // i moves in the inner loop
    //...
}
```

Listing 6: final_code.c

2.2 Parallelization with OpenMP

The optimized version introduces OpenMP directives to parallelize both the computation and the error reduction step:

```
#pragma omp parallel for reduction(max:error) collapse(2)
for (int j = 1; j < n - 1; j++) {
    for (int i = 1; i < m - 1; i++) {
        // Laplace update + error calculation
    }
}
```

Listing 7: final_code.c – OpenMP parallel loop

2.3 Division Replacement

Replacing divisions with multiplications improves performance by avoiding expensive floating-point division operations:

```
Anew[j*m + i] = (A[j*m + i + 1] + A[j*m + i - 1] +
                 A[(j-1)*m + i] + A[(j+1)*m + i]) / 4.0;
```

Listing 8: initial_code.c

```
Anew[j*m + i] = (A[j*m + i + 1] + A[j*m + i - 1] +
                 A[(j-1)*m + i] + A[(j+1)*m + i]) * 0.25f;
```

Listing 9: final_code.c

2.4 Index Precomputation

Avoiding repeated multiplications inside the inner loop reduces instruction count:

```
int idx = j * m + i;
Anew[idx] = 0.25f * (A[idx + 1] + A[idx - 1] +
                     A[idx - m] + A[idx + m]);
```

Listing 10: final_code.c – precomputed indices

3 Conclusion

The initial execution time was about 90 seconds. Through these different optimizations, while keeping the same initial problem values, we managed to get it down to 0.6 seconds.