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:

```c
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.

```c
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.

```c
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:

```c
#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 (Cache is reused since fetches bring contiguous blocks of memory). This makes a big difference because the first scenario generates the most possible amount of cache misses, and reordering the loops generates the most hits.

```c
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

```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:

```c
#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:

```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]) / 4.0;
```

Listing 8: initial_code.c

```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:

```c
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 95 seconds. Through these different optimizations, while keeping the same initial problem values, we managed to get it down to 0.61 seconds which is 157 times faster.

Extra things to notice :

- CPU used went from 1 to 5.8
- Instruction throughput went from around 650M/s to 14G/s

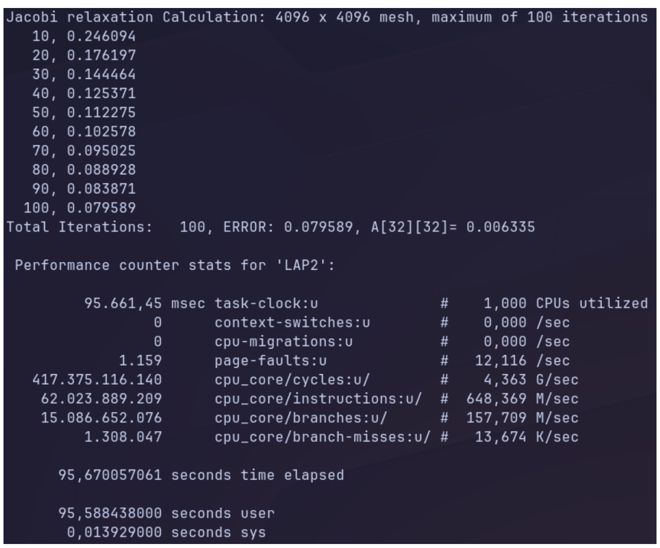
```
Jacobi relaxation Calculation: 4096 x 4096 mesh, maximum of 100 iterations
   10, 0.246094
   20, 0.176197
   30, 0.144464
   40, 0.125371
   50, 0.112275
   60, 0.102578
   70, 0.095025
   80, 0.088928
   90, 0.083871
  100, 0.079589
Total Iterations:   100, ERROR: 0.079589, A[32][32]= 0.006335

 Performance counter stats for 'LAP2':

         95.661,45 msec task-clock:u              #    1,000 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
             1.159      page-faults:u             #   12,116 /sec
   417.375.116.140      cpu_core/cycles:u/        #    4,363 G/sec
    62.023.889.209      cpu_core/instructions:u/  #  648,369 M/sec
    15.086.652.076      cpu_core/branches:u/      #  157,709 M/sec
         1.308.047      cpu_core/branch-misses:u/ #   13,674 K/sec

      95,670057061 seconds time elapsed

      95,588438000 seconds user
       0,013929000 seconds sys
```

Figure 1: Initial version runtime

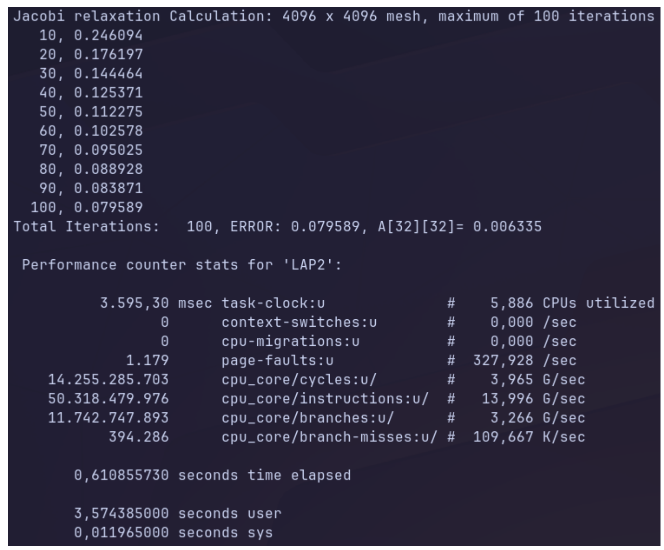
```
Jacobi relaxation Calculation: 4096 x 4096 mesh, maximum of 100 iterations
   10, 0.246094
   20, 0.176197
   30, 0.144464
   40, 0.125371
   50, 0.112275
   60, 0.102578
   70, 0.095025
   80, 0.088928
   90, 0.083871
  100, 0.079589
Total Iterations:   100, ERROR: 0.079589, A[32][32]= 0.006335

 Performance counter stats for 'LAP2':

          3.595,30 msec task-clock:u              #    5,886 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
             1.179      page-faults:u             #  327,928 /sec
    14.255.285.703      cpu_core/cycles:u/        #    3,965 G/sec
    50.318.479.976      cpu_core/instructions:u/  #   13,996 G/sec
    11.742.747.893      cpu_core/branches:u/      #    3,266 G/sec
           394.286      cpu_core/branch-misses:u/ #  109,667 K/sec

       0,610855730 seconds time elapsed

       3,574385000 seconds user
       0,011965000 seconds sys
```

Figure 2: Optimized version runtime