🎯 ALGOKOM Interactive Demo C Backend

📚 Fibonacci Methods Explained 🎨 Bilinear Interpolation Details

📊 Fibonacci Calculator - OpenMP & OpenCilk Parallel Computing

🚀 **Powered by C, OpenMP, & OpenCilk!**
Program ini menggunakan backend C dengan **dua parallel computing models**: OpenMP (GCC) dan OpenCilk (work-stealing scheduler). Pilih OpenMP saja, OpenCilk saja, atau jalankan keduanya sekaligus untuk melihat perbandingan performanya.

Enter n (0-45):

35

Mode:

🔶 OpenMP + 🔷 OpenCilk (run both)

Calculate OpenMP + OpenCilk (C Backend)

▼ 📜 Lihat Kode C Backend (Fibonacci OpenMP & OpenCilk)

Kompilasi: `gcc-15 -fopenmp -DUSE_OPENMP -o fib_omp_json fibonacci_json.c`, `<opencilk-clang> -fopencilk -DUSE_CILK -o fib_json_cilk fibonacci_json.c`.

1) Baseline & utilitas

```
int fib_sequential(int n) {
  if (n < 2) return n;
  return fib_sequential(n - 1) + fib_sequential(n - 2);
}
```

2) OpenMP

```
#ifdef USE_OPENMP
#include <omp.h>
#define CUTOFF 20

int fib_omp_task(int n) {
  if (n < 2) return n;
  if (n < CUTOFF) return fib_sequential(n);
  int x, y;
  #pragma omp task shared(x) { x = fib_omp_task(n - 1); }
  y = fib_omp_task(n - 2);
  #pragma omp taskwait
  return x + y;
}

int fibonacci_openmp_parallel(int n) {
  int res = 0;
  #pragma omp parallel
  { #pragma omp single { res = fib_omp_task(n); } }
  return res;
}
#endif
```

3) OpenCilk

```
#ifdef USE_CILK
#include <cilk/cilk.h>
#define CILK_CUTOFF 20

int fib_cilk_task(int n) {
  if (n < 2) return n;
```

```
  if (n < CILK_CUTOFF) return fib_sequential(n);
  int x = cilk_spawn fib_cilk_task(n - 1);
  int y = fib_cilk_task(n - 2);
  cilk_sync;
  return x + y;
}

int fibonacci_cilk_parallel(int n) { return fib_cilk_task(n); }
#endif
```

Results from C Program:

1. Pure Sequential (Baseline) C

Result	Time	Speedup	Efficiency
9227465	**44.609 ms**	**1x**	**100%**

2. OpenMP Parallel (Task-Based) OpenMP

Result	Time	Speedup	Efficiency	Threads	Cutoff
9227465	**10.584 ms**	**4.21x faster ⚡**	**52.69%**	**8**	**20**

Model
Fork-Join with Task Dependency

3. Cilk Parallel (Work-Stealing) Cilk

Result	Time	Speedup	Cutoff	Model
9227465	**5.976 ms**	**7.46x faster ⚡**	**20**	**Work-Stealing Scheduler**

📊 **Performance Summary:**
Cilk Parallel is **7.46x faster** than Sequential.
Time delta: **86.6%** (38.63 ms)
Computed Fibonacci(35) = **9227465**

💡 **Available Implementations:**

🔶 **OpenMP Mode:** Menggunakan GCC dengan fork-join task model
• Sequential: Baseline single-threaded
• OpenMP Serial: Framework tanpa parallelization (overhead measurement)
• OpenMP Parallel: Multi-threaded dengan task parallelism

🔷 **OpenCilk Mode:** Menggunakan OpenCilk compiler dengan work-stealing scheduler
• Cilk Serial: Single-threaded tanpa spawn
• Cilk Parallel: Work-stealing task parallelism

� **Key Metrics:**
• **Speedup**: Waktu Sequential / Waktu Parallel
• **Efficiency**: (Speedup / Jumlah Thread) × 100%
• **Cutoff**: Threshold untuk menghindari overhead task creation

📚 Parallel Computing Concepts

Sequential Execution

Cara Kerja: Satu thread menghitung semua secara berurutan
Kelebihan: Simple, predictable, no overhead
Kelemahan: Tidak memanfaatkan multi-core processor

OpenMP Serial

Cara Kerja: Menggunakan OpenMP framework tapi tanpa task parallelism
Tujuan: Mengukur overhead dari OpenMP runtime

Overhead: Biasanya 0-2% (sangat minimal)

OpenMP Parallel (Task-Based)

Cara Kerja: Fork-Join model dengan task dependency
Model: #pragma omp task untuk spawn parallel tasks
Cutoff Strategy: Sequential untuk n < 20 (menghindari overhead)
Keunggulan: Load balancing otomatis, skalabilitas baik

Cilk Parallel (Work-Stealing)

Cara Kerja: Spawn tasks dengan work-stealing scheduler
Model: cilk_spawn untuk spawn parallel tasks
Synchronization: cilk_sync untuk wait tasks
Cutoff Strategy: Sequential untuk n < 20 (menghindari overhead)
Work-Stealing Scheduler: Idle threads mencuri task dari busy threads
Keunggulan: Optimal load balancing, cache-friendly, minimal overhead
Formula: $T_P \approx W/P + D$ (W=total work, P=processors, D=depth)

🎨 Bilinear Interpolation - Image Upscaling (C + OpenMP)

📖 Bilinear Interpolation untuk Image Upscaling

Bilinear Interpolation adalah teknik image processing untuk memperbesar (zoom) gambar dengan mengestimasi nilai pixel baru berdasarkan 4 pixel terdekat dari image original. Hasilnya lebih smooth dibanding Nearest Neighbor.

🚀 **Powered by C & OpenMP!**
Program C backend melakukan real image resizing dengan bilinear interpolation menggunakan OpenMP untuk parallel processing.

Image File:

Gantry Crane (Default)

Scaling Factor (1.5x - 4.0x):

2.0x

🚀 Process with C Backend (Serial & Parallel Auto)

Image Processing Results:

Image Processing Results C OpenMP

Original Size	Upscaled Size	Scaling Factor	Total Pixels
400×264	**800×528**	**2.00x**	**0.42M**

Serial Processing (Baseline) C Serial

Execution Time	Throughput
3.200 ms	**132.00 Mpixels/s**

2-Thread Parallel OpenMP

Execution Time	Speedup	Efficiency	Throughput
14.800 ms	**4.63x slower**	**10.81%**	**28.54 Mpixels/s**

4-Thread Parallel OpenMP

Execution Time	Speedup	Efficiency	Throughput
14.000 ms	**4.38x slower**	**5.71%**	**30.17 Mpixels/s**

8-Thread Parallel OpenMP

Execution Time	Speedup	Efficiency	Throughput
19.800 ms	**6.19x slower**	**2.02%**	**21.33 Mpixels/s**

💡 **Memory-Bound Algorithm:**
Bilinear interpolation is **memory-bound**, not compute-bound.
Speedup limited by memory bandwidth, not number of CPU cores.
This is why parallel versions may show low efficiency on small images.

Original Image

400×264 px

Serial Result

Serial: 800×528 px, 3.20 ms

Parallel Result (8-Thread)

Parallel (8T): 800×528 px, 19.80 ms

▼ 📜 Lihat Kode C Backend (Serial & OpenMP Parallel)

Program dijalankan: `./bilinear <file> [scale]` (default scale 2.0, mengikuti slider di UI). Cuplikan ini sinkron dengan `bilinear_serial_parallel.c`.

1) bilinear_interpolate

```
static inline double bilinear_interpolate(double x, double y,
    double Q11, double Q21, double Q12, double Q22) {
  double fx1 = Q11 + (Q21 - Q11) * x;
  double fx2 = Q12 + (Q22 - Q12) * x;
  return fx1 + (fx2 - fx1) * y;
}
```

Menggabungkan 4 tetangga (Q11, Q21, Q12, Q22) dengan fraksi jarak x,y (fx, fy).

2) bilinear_resize_serial

```
unsigned char** bilinear_resize_serial(
    unsigned char** src, int src_h, int src_w,
    int new_h, int new_w) {
  unsigned char** dst = malloc(new_h * sizeof(unsigned char*));
  for (int i = 0; i < new_h; i++) dst[i] = malloc(new_w * 3);

  double rx = (double)(src_w - 1) / (new_w - 1);
  double ry = (double)(src_h - 1) / (new_h - 1);

  for (int i = 0; i < new_h; i++) {
    for (int j = 0; j < new_w; j++) {
      double sx = j * rx, sy = i * ry;
      int x1 = (int)sx, y1 = (int)sy;
      if (x1 >= src_w - 1) x1 = src_w - 2;
      if (y1 >= src_h - 1) y1 = src_h - 2;
      int x2 = x1 + 1, y2 = y1 + 1;
      double dx = sx - x1, dy = sy - y1;
      for (int c = 0; c < 3; c++) {
        double Q11 = src[y1][x1 * 3 + c], Q21 = src[y1][x2 * 3 + c];
        double Q12 = src[y2][x1 * 3 + c], Q22 = src[y2][x2 * 3 + c];
        double val = bilinear_interpolate(dx, dy, Q11, Q21, Q12, Q22);
```

```
        dst[i][j * 3 + c] = (unsigned char)(val + 0.5);
      }
    }
  }
  return dst;
}
```

Loop baris-kolom: mapping dest→src, clamp batas, hitung dx/dy, lalu blend per channel RGB menggunakan 4 tetangga.

3) bilinear_resize_parallel (OpenMP)

```
unsigned char** bilinear_resize_parallel(
    unsigned char** src, int src_h, int src_w,
    int new_h, int new_w, int threads) {
  unsigned char** dst = malloc(new_h * sizeof(unsigned char*));
  for (int i = 0; i < new_h; i++) dst[i] = malloc(new_w * 3);

  double rx = (double)(src_w - 1) / (new_w - 1);
  double ry = (double)(src_h - 1) / (new_h - 1);

  omp_set_num_threads(threads);
  #pragma omp parallel for collapse(2) schedule(dynamic)
  for (int i = 0; i < new_h; i++) {
    for (int j = 0; j < new_w; j++) {
      double sx = j * rx, sy = i * ry;
      int x1 = (int)sx, y1 = (int)sy;
      if (x1 >= src_w - 1) x1 = src_w - 2;
      if (y1 >= src_h - 1) y1 = src_h - 2;
      int x2 = x1 + 1, y2 = y1 + 1;
      double dx = sx - x1, dy = sy - y1;
      for (int c = 0; c < 3; c++) {
        double Q11 = src[y1][x1 * 3 + c], Q21 = src[y1][x2 * 3 + c];
        double Q12 = src[y2][x1 * 3 + c], Q22 = src[y2][x2 * 3 + c];
        double val = bilinear_interpolate(dx, dy, Q11, Q21, Q12, Q22);
        dst[i][j * 3 + c] = (unsigned char)(val + 0.5);
      }
    }
  }
```

```
  return dst;
}
```

Sama seperti serial, tetapi loop baris-kolom di-parallel-kan dengan OpenMP (`collapse(2)`, `schedule(dynamic)`) dan jumlah thread diatur via `omp_set_num_threads`.

📚 How It Works

1. **Read PNG Image** - Load original image via ImageMagick
2. **Bilinear Interpolation** - For each new pixel, interpolate from 4 nearest neighbors
3. **Serial vs Parallel** - Compare performance between single-threaded and multi-threaded versions
4. **Memory-Bound Algorithm** - Speedup limited by memory bandwidth, not CPU cores
5. **Output** - Save upscaled image as PPM/PNG