Parallelized Genetic Algorithms:

Graph Partition Problem Case Study

Presented By: Jason Fries, Jim Paton

22C:196 Parallel Programming with Messages, Threads and GPUs

Overview

- Graph Partition Problem
- Genetic Algorithms (GA)
- OpenMP GA Implementation
- CUDA GA Implementation
- Questions

Graph Partition Problem

- Given a graph G = (V, E)
- Find a partitioning Π of G with minimal cost
- Typical cost function:

$$C(\mathrm{P}) = \sum_{(u,v) \in V, \mathrm{P}(u) \neq \mathrm{P}(v)} w[(u,v)] + c \arg\max_{i,j \in \mathrm{P}} (|i| - |j|)$$

- NP-complete for C(Π) given 2 partitions

Restrictions on solution

- Specific bounds on number of partitions
- Partitions must be balanced
- Imbalance of partitions is limited
- Vertices are weighted

Applications

- Designing VLSI circuits
- Routing in distributed systems
- Image segmentation in computer vision
- Virtual memory paging systems
- Mapping parallel programs on parallel architectures

(from Talbi and Bessiere)

Genetic Algorithms

- A stochastic optimization heuristic
- Typically used in search space problems
- Utilizes iterative natural selection and recombination (akin to genetic evolution) via these operations:
  - Crossover
  - Mutation
  - Selection

Encoding Chromosomes

- Candidate solutions are encoded as strings, commonly called chromosomes. **A1 = 0 1 1 0 1 1 1 1**
- Population individuals are initially assigned a random chromosome (or one defined by some qualitative heuristic)

Crossover

Two individuals "mate" and exchange chromosomes.

For string A1 and A2:

A1 = 0 1 1 0 1 1 1 1

A2 = 1 1 1 0 0 0 0 0

There are length-1 crossover points. Randomly chose a crossover point and recombine:

A1 = 0 1 1 0 | 0 0 0 0

A2 = 1 1 1 0 | 1 1 1 1

Mutation

Secondary operator used to restore genetic diversity.

For string A1:

A1 = 0 1 1 0 1 1 1 1

Randomly alter some value (example: flip bit 2):

A1 = 0 0 1 0 1 1 1 1

Selection

Solution populations are evaluated according to some fitness function. The best members are selected for mating (i.e., crossover)

Bad traits are eliminated from the pool, as those members do not survive this step.

Mating can occur across the whole population or subsets, depending on the class of GA

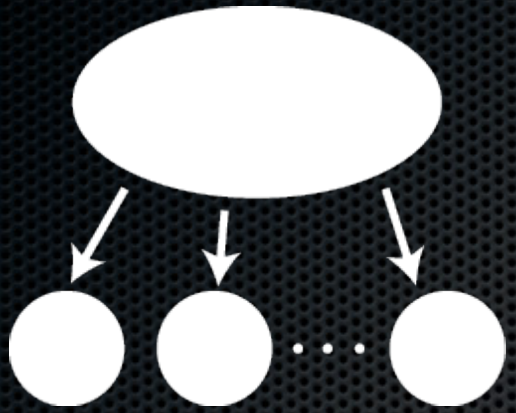
Why do GAs work?

- Most widely accepted answer: **Holland's Schema Theorem**
  - *"A schema is a template that identifies a subset of strings with similarities at certain string positions"*
  - **Example:** binary string 1*0*1 defines a set of strings 5 bits long with
    - *"1"* at fixed positions **1,5**
    - *"0"* at fixed position **3**
    - *"1"* or *"0"* at wildcard positions **2,4**

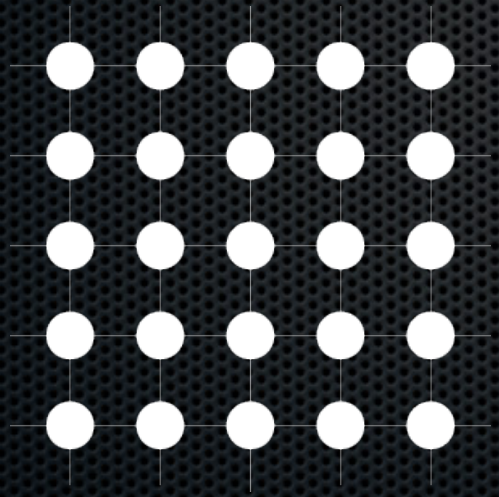
Why do GAs work?

- Schema fitness is the average fitness of all matching strings.
- Theorem: Short, low-order, schemata with above-average fitness increase exponentially in successive generations.

Classes of GAs

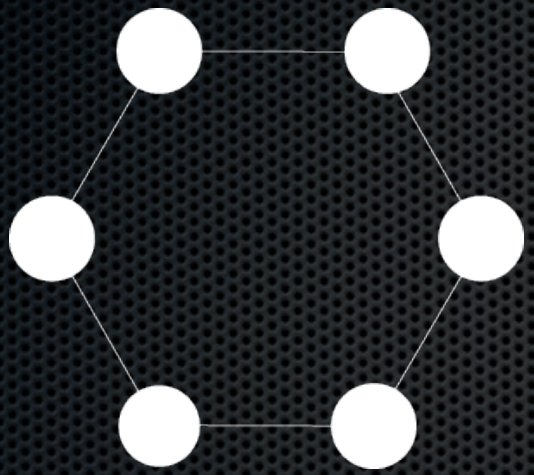


- **Single Population (Master/Slave)**
  - Fitness evaluated across processors. Master assigns individuals to slaves.



- **Single Population (Fine Grained)**
  - Selection & mating limited to spatially structured subregions of the population (“neighborhoods”)

Classes of GA



- **Multiple Populations (Coarse Grained)**
  - Individuals exchanged infrequently between subpopulations ("migration")
  - Most popular GA implementation
  - Coarse grained GAs converge quickly, but solutions tend to converge at local optima.

Open Theoretical Questions

- Selecting population size is tricky and depends on the problem.
- Panmictic (single population where all members can be partners) GAs find better solutions at the expense of more time.
- Migration rates in coarse grained GAs induce panmictic solution quality while preserving speedup.
- However it is unknown if there is a theoretically optimal migration rate.

OpenMP implementation

- Serial CPU and OpenMP implementation share source files
- Cost function

$$C(\mathrm{P}) = \sum_{(u,v)\in V,\mathrm{P}(u)\neq\mathrm{P}(v)} w[(u,v)] + c \arg\max_{i,j\in\mathrm{P}}(||i| - |j||)$$

  with c = 100, k = 2
- Fitness = -cost
- Block-based

Algorithm

- Generate a random graph of 500 vertices
- Generate a number of random individuals (100)
- For each generation (1000 generations total):
  - Quick sort by fitness
  - Select top 10%
  - Reproduce remaining individuals through crossover
  - Mutate 5% of resulting individuals

Model definitions

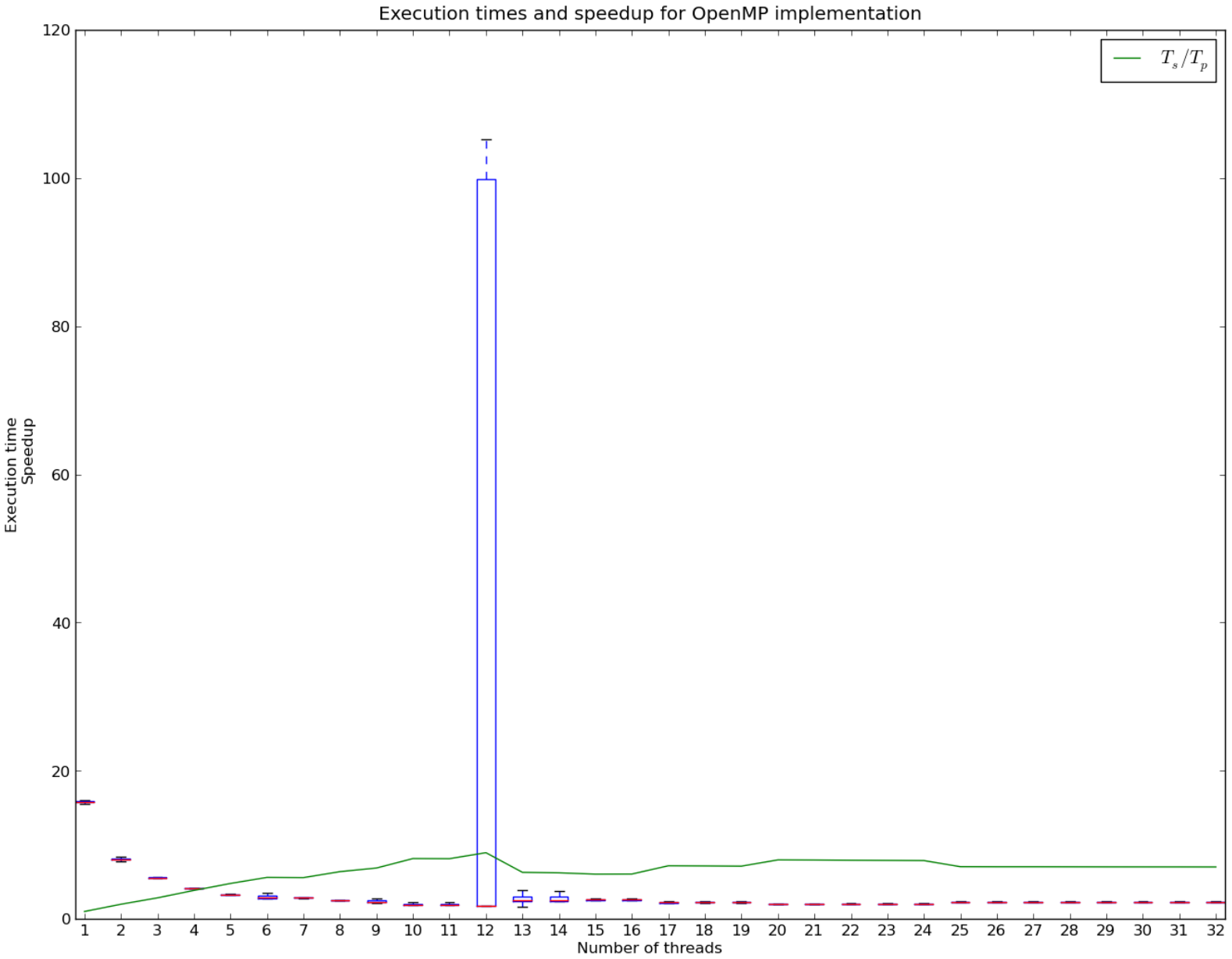
```
typedef uint8_t Chromosome; // individual.chromosomes[i] = the
partition number of element i

typedef struct individual {
     Chromosome * chromosomes; // array of chromosomes
     size_t size; // number of chromosomes
} Individual;

typedef Graph Model;
```

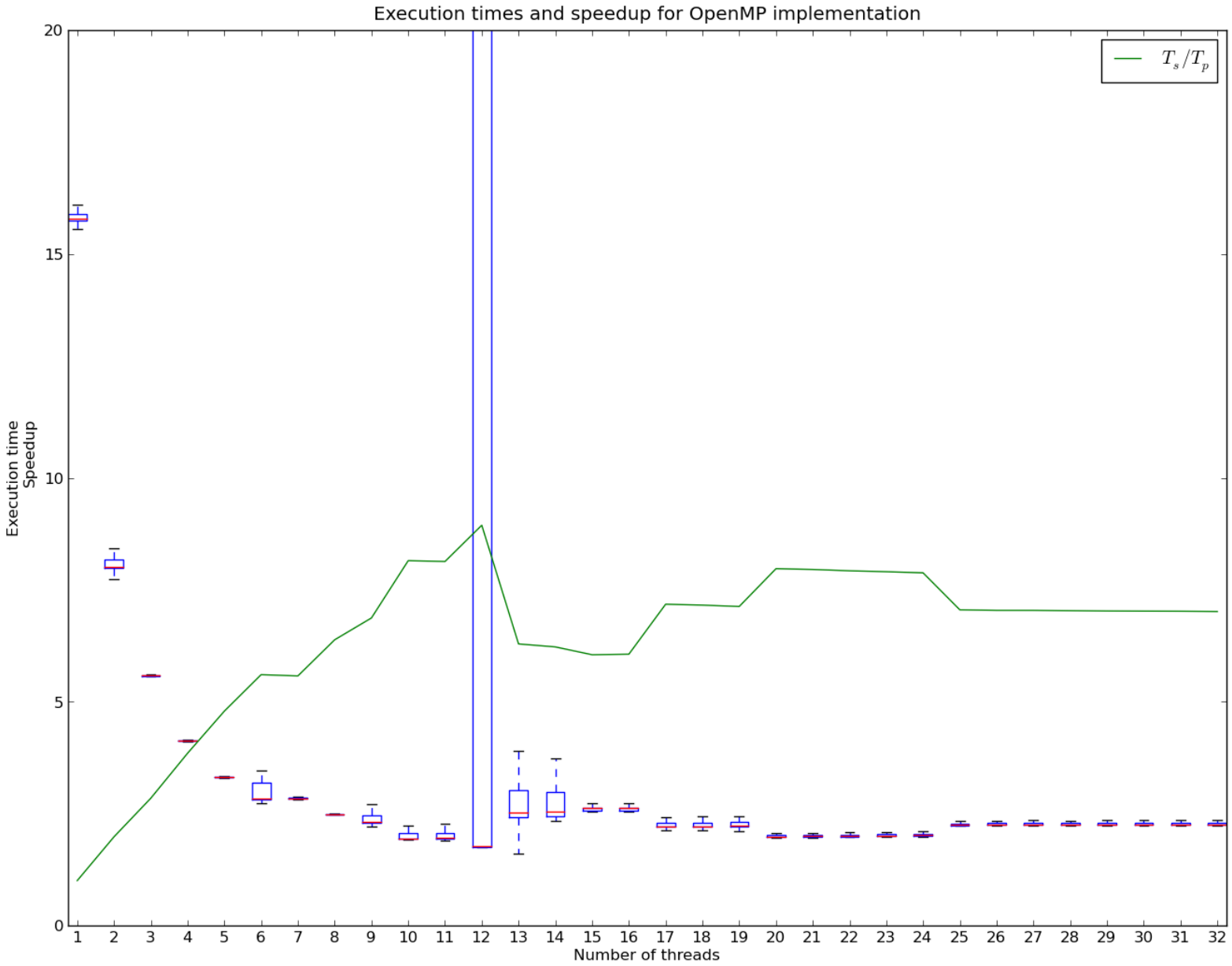
Random individual generation

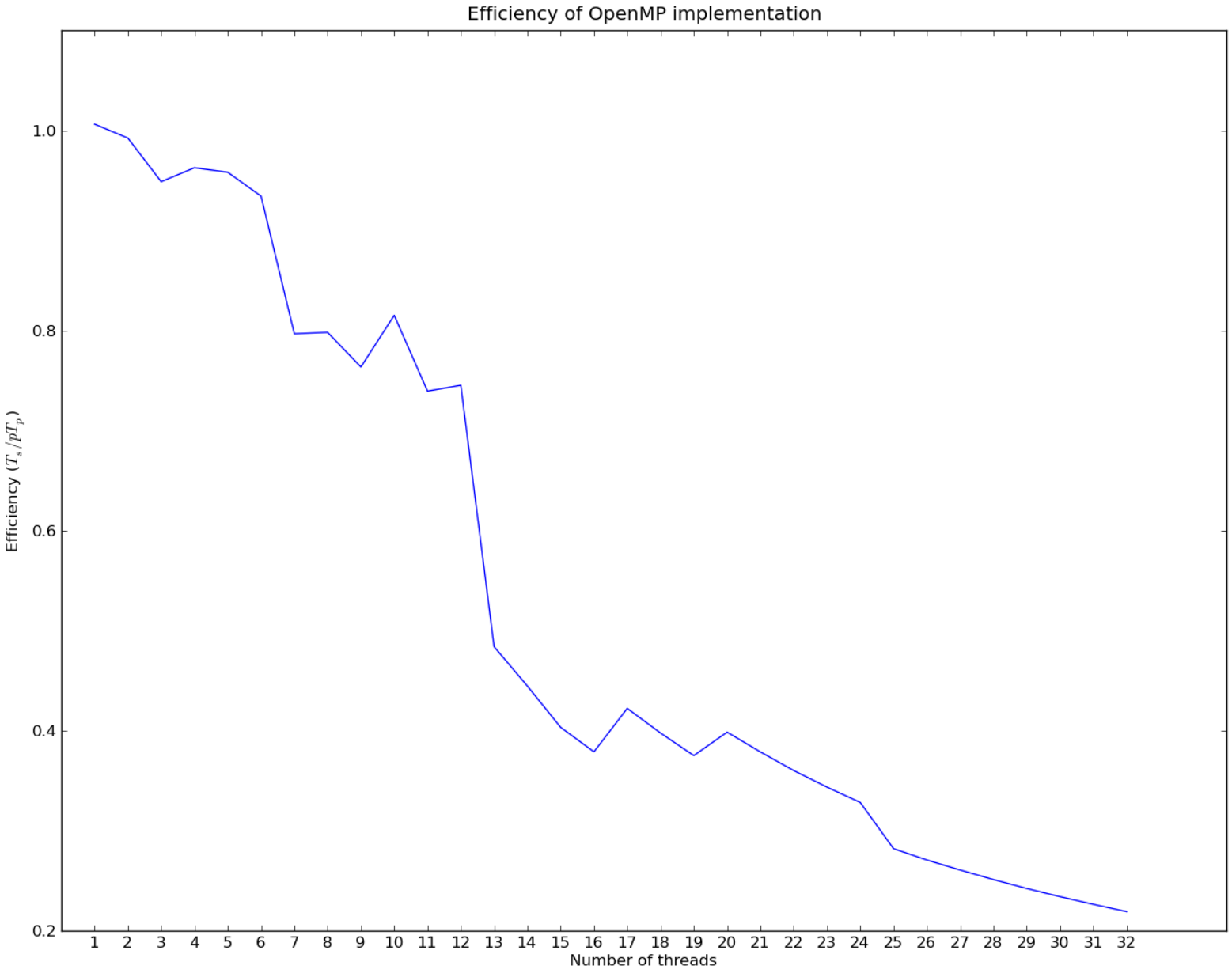
```
Individual random_individual( Graph * g, int k ) {
    Individual individual = make_Individual( g->num_vertices );

    /* create a list of vertices */
    int vertices[g->num_vertices];
    /* initialize them and shuffle using Fisher-Yates inside-out
version */
    vertices[0] = 0;
    for (int i = 1; i < g->num_vertices; i++) {
     int j = rand() % (i + 1);
     vertices[i] = vertices[j];
     vertices[j] = i;
    }

    /* deal out vertices to partitions */
    for (int i = 0; i < g->num_vertices; i++) {
     individual.chromosomes[vertices[i]] = i % k;
    }

    return individual;
}
```

Parallel fitness evaluation

```
    float max_fitness = -FLT_MAX;
    #ifndef SERIAL
    int num_per_block = generation_size / num_blocks;
    int i, j;
    #pragma omp parallel for shared(max_fitness, best, k,
k_penalty, g) private(i, j)
    for (j = 0; j < num_blocks; j++) {
        for (i = j * num_per_block; i < min((j + 1) *
num_per_block, generation_size); i++) {
    #else
        for (int i = 0; i < generation_size; i++) {
    #endif
            fitnesses[i] = fitness(&population[i], g, k,
k_penalty);
            #ifndef SERIAL
            #pragma omp critical
            {
            #endif
            if (fitnesses[i] > max_fitness) {
                max_fitness = fitnesses[i];
                if (max_fitness > best_fitness) {
                    best = copy(&population[i]); // save a
copy
                    best_fitness = max_fitness;
                }
            }
        #ifndef SERIAL
        }
        #endif
    }
    #ifndef SERIAL
    }
    #endif
```


Execution times and speedup for OpenMP implementation
T_s/T_p
Execution time
Speedup
Number of threads
0
20
40
60
80
100
120
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32


Execution times and speedup for OpenMP implementation
T_s/T_p
Execution time
Speedup
Number of threads
0
5
10
15
20
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32


Efficiency of OpenMP implementation
Efficiency (T_s/pT_p)
1.0
0.8
0.6
0.4
0.2
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Number of threads

CUDA implementation

Extends OpenMP implementation

Individuals → Blocks of chromosomes

Fitness function implemented as a kernel

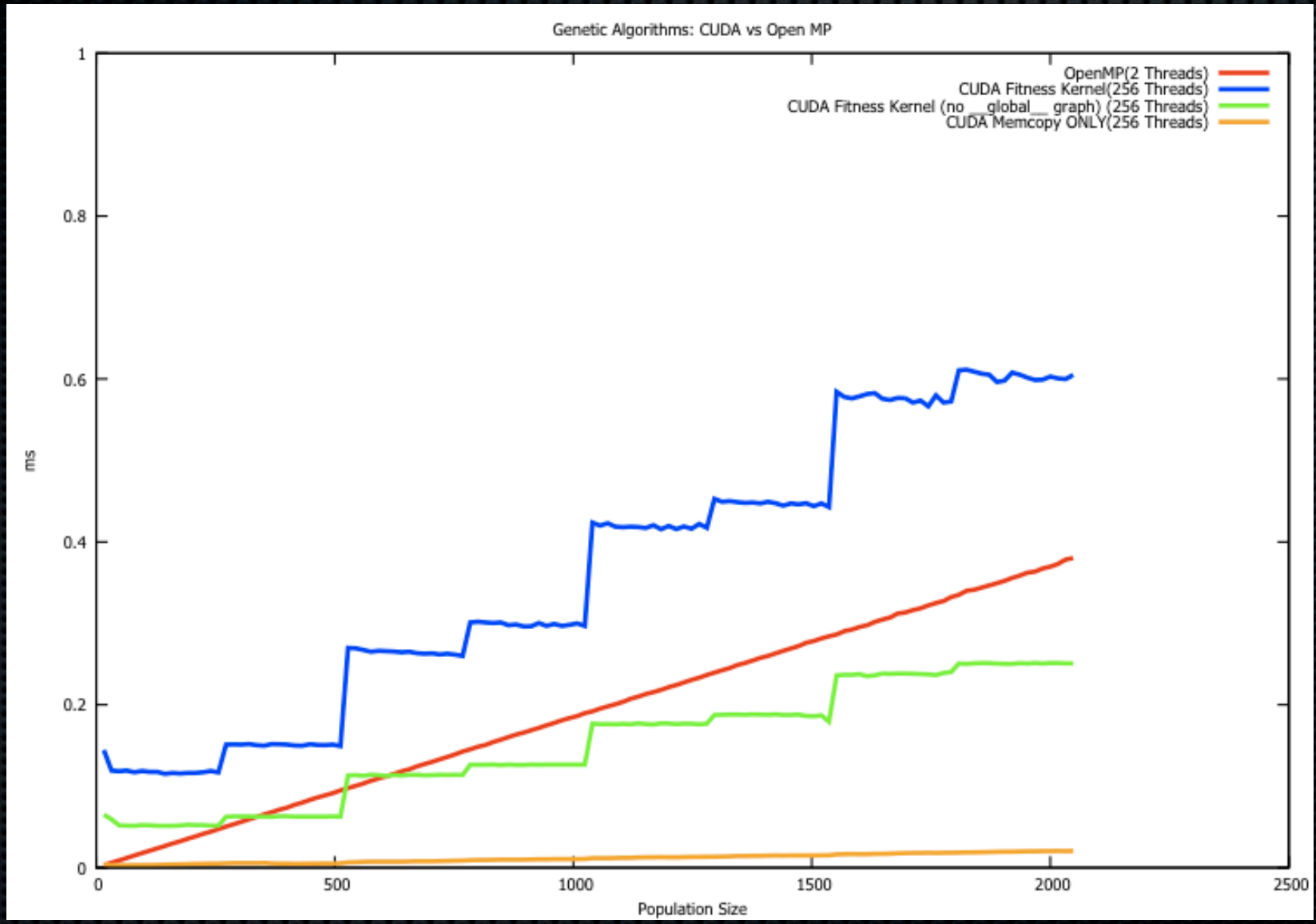
Fitness function must be implemented entirely on the GPU!

```
__global__
void gpu_fitness_v1( Chromosome *population,
                     Edge *edges,
                     float *fitnesses,
                     int num_edges,
                     int num_vertices,
                     size_t *counts,
                     size_t k,
                     float k_penalty ) {

    //shared memory doesn't really seem to help speed this up.
    //extern __shared__ float s_fitnesses[];

    float fitness = 0.0;
    int i,tid,offset;

    tid = blockIdx.x * blockDim.x + threadIdx.x;
    offset = tid * num_vertices;

    // NOTE: Edges could be stored in __constant__ memory, as we don't write to them
    for (i = 0; i < num_edges; i++) {

        if( population[ edges[i].v0 + offset ] != population[ edges[i].v1 + offset ] ){
            fitness -= edges[i].weight;
        }
    }

    //clear counts
    for(i=0; i < k; i++){
        counts[ tid * k + i ] = 0;
    }

    for (i = 0; i < num_vertices; i++)
        counts[ (int)population[i+offset] + tid * k ]++;

    // find count difference penalty
    size_t min, max;
    max = 0;
    min = SIZE_MAX;
    for (i = 0; i < k; i++) {
        if (counts[tid * k + i] < min) min = counts[tid * k + i];
        if (counts[tid * k + i] > max) max = counts[tid * k + i];
    }

    fitness -= (float)(max - min) * k_penalty;

    __syncthreads();

    fitnesses[tid] = fitness;

}
```

But implementation with fitness kernel and CPU crossover/mutate is slower than OpenMP !

Why?

Population Size: Performance



Bottleneck occurs for two primary reasons:

- Fitness values must be copied off device every generation
- Graph data and population data access, simultaneously cause memory collisions

Collisions actually have larger impact on performance (note green line on previous slide)

GPU kernels for Crossover and Mutate have little impact – the fitness evaluation takes up most of the time.

How to fix?
Move graph to constant memory introduces more size constraints (max 65536kb).

Perhaps to divide graph into shared blocks of fast __shared__ mem and cache?

Questions?

References

- Cantú-Paz, E. A Survey of Parallel Genetic Algorithms
- Holland's schema theorem, http://en.wikipedia.org/wiki/Holland's_schema_theorem