

15-418 Project Report

Cliff Zhu (zhaoxiz), Rosie Sun (weijias1)

[Github Repo](#)

[Project Proposal](#)

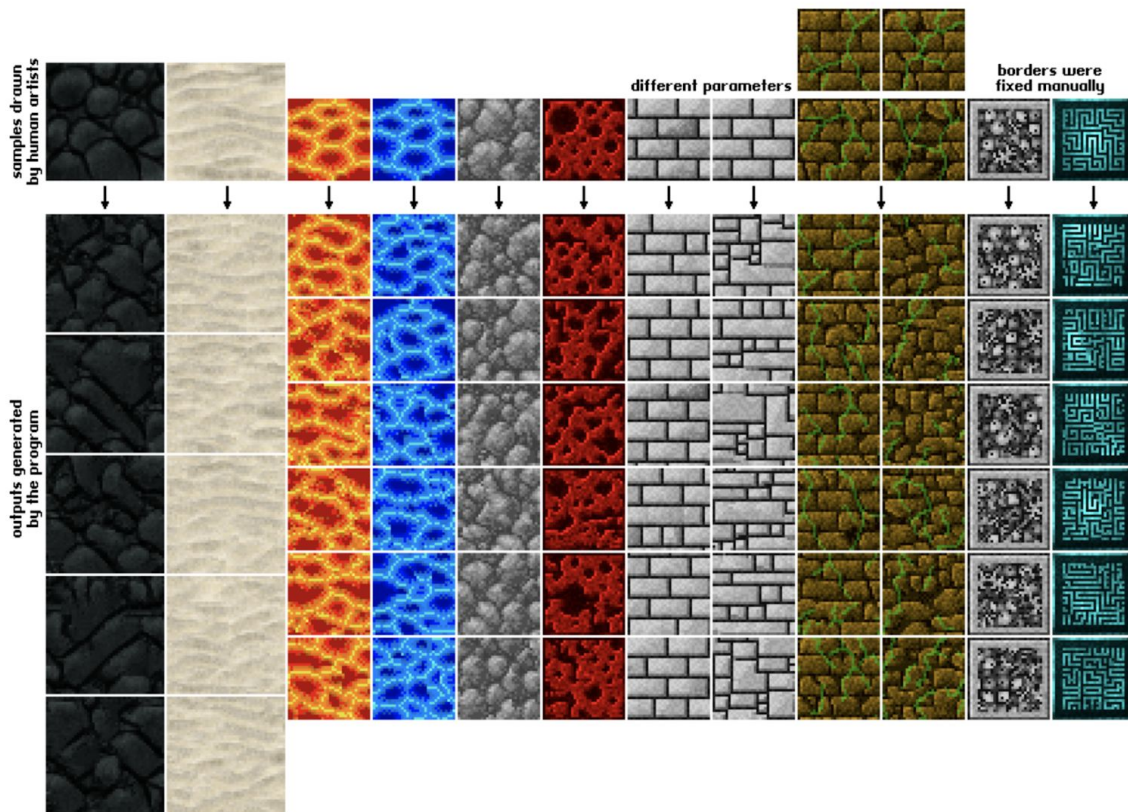
[Project Website](#)

Project Summary

Texture synthesis is an image-generation technique commonly used to produce high-quality textures when rendering synthetic graphics. In this project we used multithreading with OpenMP to parallelize an algorithm called **non-parametric sampling** for texture synthesis.

Background

Define texture as some visual pattern of an infinite 2D plane. Texture synthesis is the process of taking a finite fixed shape sample from a texture to generate other samples of a different dimension from the given texture. Potential graphic applications of texture synthesis include image de-noising, occlusion fill-in, image compression, etc. Below are some examples of inputs and respective sample outputs of texture synthesis. The first row represents the input (a fixed n by n sample), the bottom rows are the output generated by the program.



We present a high-level description below of the sequential algorithm proposed in the paper **Texture Synthesis by Non-parametric Sampling** by Alexei A. Efros and Thomas K. Leung of UC Berkeley.

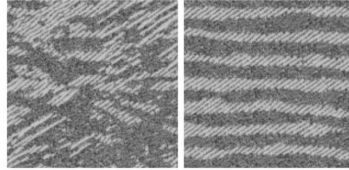
Define:

- Neighborhood of p , $\omega(p)$, to be a $N \times N$ square window centered at pixel p
 - The size of the window N is a free parameter that allows the user to specify how stochastic the texture should be. Below the result on the left is generated with a small window size N that gives a high stochastic image, while a larger value of N yields a more uniform synthesis on the right.

Sample



Synthesis results



- $d(\omega_1, \omega_2)$ to be the perceptual distance between any two neighborhoods. Smaller distance means that two neighborhoods have higher similarity.
 - To find the distance, the normalized sum of squared differences (SSD) approach is taken with slight optimization. The SSD approach gives the same weight any mismatched pixels regardless of their location. Since we would like to preserve the local structure of the original texture, a Gaussian kernel is used to assign less error weight to the edge pixels.

Algorithm:

```

For each pixel  $p$  in image:
  For all neighborhoods  $\omega$  in the given sample:
    • Compute  $d(\omega, \omega(p))$  based on modified SSD approach
    • Let  $\omega_{\min}$  to be the neighborhood with minimal distance
      (highest similarity) to  $\omega(p)$ .
    • Let  $S = \{ \omega : d(\omega, \omega(p)) \leq (1 + \epsilon) * \omega_{\min} \}$  be the set of
      neighborhoods in sample with similarity above some threshold.
  Randomly select a neighborhood  $\omega'$  from  $S$ .
  Color current pixel  $p$  with the color of the center pixel of  $\omega'$ 

```

Both input and output samples are represented by 2D arrays of pixels. Each pixel is encoded via RGB. Define the input sample size to be sw by sh . Besides the input sample, two parameters are passed in: window size and radius. Window size is already defined above. The radius specifies how large the output would be. Specifically, the output width/height is calculated by $2 * \text{radius} + \text{window}$.

There are three intermediate 2D arrays.

- The flag array (output_width by output_height) marks if the pixel value of each position of the output texture is computed.
- The kernel array (window_size by window_size) computes and stores the gaussian kernel of each of the neighborhood centered at each position of the output texture
- The distance array (sw-window_size+1 by sh-window_size+1) computes the distance between any our neighborhood of interest and all the possible neighborhoods in the sample.

The part of the algorithm that is most computationally involved is the part of computing the distance between our neighborhood of interest and other neighborhoods to determine if the distance is within a threshold. The reason for this is we need to consider all the neighborhood of the sample every time for a target neighborhood. When calculating distance, we are looping over each pixel within the window_size by window_size neighborhood. The part of the code can hugely benefit from parallelization because there is almost no dependency. Each neighborhood distance computation can be performed independently. This part is data parallel. The only critical section is where we need to find the minimum of all the computed distances.

For locality, we have both spatial and temporal locality. For spatial locality, we are going from the top left neighborhood in a row major order down to the bottom right neighborhood. Within each neighborhood, we are also preceding in a row major way one pixel by pixel. This is analogous to the block iteration approach. For temporal locality, we are frequently accessing the same pixel as once we access a neighborhood, at least the next window_size neighborhoods would share similar pixels.

Approach

We tried to mainly parallelize the function [synthesize](#) with **OpenMP**, which supports **multi-threaded, shared address space parallelism**.

Performance bottleneck

We first did benchmark experiments with the [original serial implementation](#) and timed different parts of the program. We observed that the performance bottleneck is computing the squared distance between the result window and all windows in the sample image. The sequential code iterates through the windows in samples one by one and generates a weighted distance value per (result, sample) pair. This operation is especially time-consuming when the difference between input window size w and sample width/height sw/sh is large. So we tried to focus our optimization in the [compute_dist](#) part of the program.

Change to data structure

We first made changes to the original serial implementation to enable better mapping to a parallel machine:

- The original version performs pixel read/write to the Image object throughout the computation. We observed these read/write operations greatly slowed down the program performance.
- In our improved serial version, we first map the image object to a 2D double array of size `[image_size][3]`, where each inner array is a list of {R, G, B} values for each pixel. In the process of `synthesize`, we perform all computations directly on the 2D pixel array, and write to the image object only when we finish all computations at the output stage. This change in data structure improves performance by allowing threads to read pixel values per result/sample window more efficiently.
- We also added timers to different parts of the program. This allows us to observe what's the performance bottleneck. .

Parallel Approach

Our key observation is that there's no dependency between the values between each sample window - computing the distance of the next `[sample_window, result_window]` pair does not depend on previously-generated values. Thus, we parallelize the for-loop to iterate through all sample windows. Each thread is responsible for computing the distance of 1 pair of windows at a time. This gave us around 4.5-5x speedup from the improved serial version.

Further optimization with various techniques

To further optimize the program performance, we tried to apply the techniques we learned throughout this semester, including:

- **Separate-accumulate:** In `compute_dist` we need to get the minimum of distances between the result window and all sample windows. In the naive approach, the updates to `min_dis` requires synchronization across threads. To eliminate synchronization, we used separate-accumulate by allocating `n_thread` values in memory. Each thread will only make updates to `scratch_vector[tid]` during computation. Finally we accumulate these min distances to obtain the overall minimum `min_dis` with openmp reduction since min operation is commutative.
- **Sub-block iteration:** the function `getDistanceOfBatch` needs to iterate through all pixel values in result and sample windows and accumulate the pairwise weighted square distances. To improve temporal locality in cache, we changed the traversal order to block the matrix into 16x16 sub-matrices. This improves performance by allowing a memory-access pattern that better fits each cache line and thus reducing the number of cache misses when loading pixel data from memory.
- **Avoid writes to memory, if not necessary:**
 - Originally we made updates to the minimum-distance vector by performing `scratch_vector[tid] = min(dist, scratch_vector[tid])`. Note that this line requires a

write to memory even when there's no need to update the minimum, inducing low arithmetic intensity.

- Then we changed it the following so we write to memory only when required: `if (dist < scratch_vector[tid]) { scratch_vector[tid] = dist; }`

- **Thread scheduling:**

Based on our experience implementing rat-simulation in assignment 3, we first tried dynamic assignment for better performance. However, we found from experiments that changing thread-scheduling to static actually improved the performance by 7-8%.

We observed that in our program, each iteration of for-loop takes roughly the same amount of time, because the number of pixel values needs to be computed per (result, sample) window is roughly equal. Therefore, static scheduling works well as they have less overhead. Whereas in rat-simulation, each iteration can have very different runtime due to the inconsistent region size and degree of connections. So dynamic schedules are more suited in that situation.

By applying the techniques above, we were able to achieve 6.5-7x speedup from the improved serial version, and an average of 27x speedup from the original serial version.

Techniques we tried but found ineffective


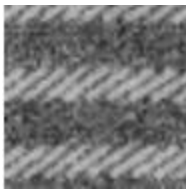
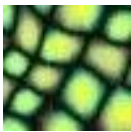
- **Batch partition schemes:** The original version of the code uses an inner-to-outer traversal scheme. Pixels in each batch will have the same distance to the center. During the optimization process we also tried to process each row as a batch. Our experiment revealed that the row-partition approach has roughly the same performance as the one with radius. This at first is unexpected since we thought the row-partition approach would yield better locality. We later found out that since during the computing of distance we are always doing a row-major order traversal and the only change is the target neighborhood, the locality would be almost the same. Comparing these two schemes, radius-partition yields a smoother and more organic synthesized output. In the end we chose radius-partition since it better preserves the original texture.
- **omp critical:** An alternative to calculate min-dis by using OpenMP reduction is to use `omp critical` to accumulate the minimum distance between the result window and all sample windows. We found this to be less effective than the built-in reduction. By using the built-in reduce rather than writing our own version of reduce with critical, we were able to raise the level of abstraction and take advantage of the optimized implementation of the functionality we desired to achieve.
- **Dynamic scheduling:** See [Thread scheduling](#) section above.

- We do recognize there are other potential sources of parallelism in the code. For example, in `find_moves` operation it's possible to parallelize the process to filter out those windows with distance greater than the threshold. However, we observed from experiments that `find_moves` runs pretty fast and only takes up 1-3% of the total runtime. The overhead to schedule multi-threading for this section would offset the speedup from parallelism, so we decided to keep it serial.

Results

Program Inputs

We selected 3 sample images that vary in their sizes and colors as program inputs. We ran benchmark experiments against different values for parameters **window size w** (determines the stochasticity of the texture) and **radius r** (determines the size of the output image).

Sample1 (30x24)	Sample2 (93x92)	Sample3 (49x48)
		

Performance Measurement

Our approach to measure the program performance is very similar to how we measured rat simulation in Assignment 3 and 4. We use `cycletimer` to record the performance of each sections of the program (unit = ms). Program breakdowns are as follows:

<code>startup</code>	start the synthesis process, allocate necessary data structure
<code>get_batch</code>	at the beginning of each batch, get all pixels in the output array that need to be processed (colored) in the current batch
<code>compute_dist</code>	for each pixel in result, compute the distances of (result_window, sample_window) pair for all possible windows in the sample and find the minimum distance
<code>find_moves</code>	based on the minimum distance, filter sample windows to include only those within threshold. Then randomly select a valid sample window and use the color of its center pixel to color the current pixel.

Benchmarks

There are 3 versions of code we used to generate measurement in benchmark experiments.

Version	Description	How we obtained results
Original sequential version	The original serial implementation on github	Run executable <code>./texturesynOrigin</code>
Improved serial version	Our improved version of the original implementation (= our program running with $t = 1$)	Run executable <code>./texturesyn</code> with $T=1$
Parallel version	Our improved version with t number of threads, where t is a user-specific parameter. In benchmark experiments we used $t = \{2, 4, 8\}$.	Run executable <code>./texturesyn</code> with $-t T$ $T = \{2, 4, 8\}$

Executable usage

```
./texturesyn -s SFILE -o OFILE -w WINDOW -r RADIUS [-t THD] [-I]
  -s SFILE          Sample file path
  -o OFILE          Output file path (no file extension required)
  -w WINDOW         Window size
  -r RADIUS         Output radius
  -t THD            Number of threads
  -I               Instrument simulation activities
  -h               Print this message
```

Example usage:

```
./texturesyn -s src/sample1.ttr -o out/sample1 -w 10 -r 100 -I -t 8
```

will run the program with 8 threads and generate an output image `sample1_10_100.ppm` in `./out/` folder.

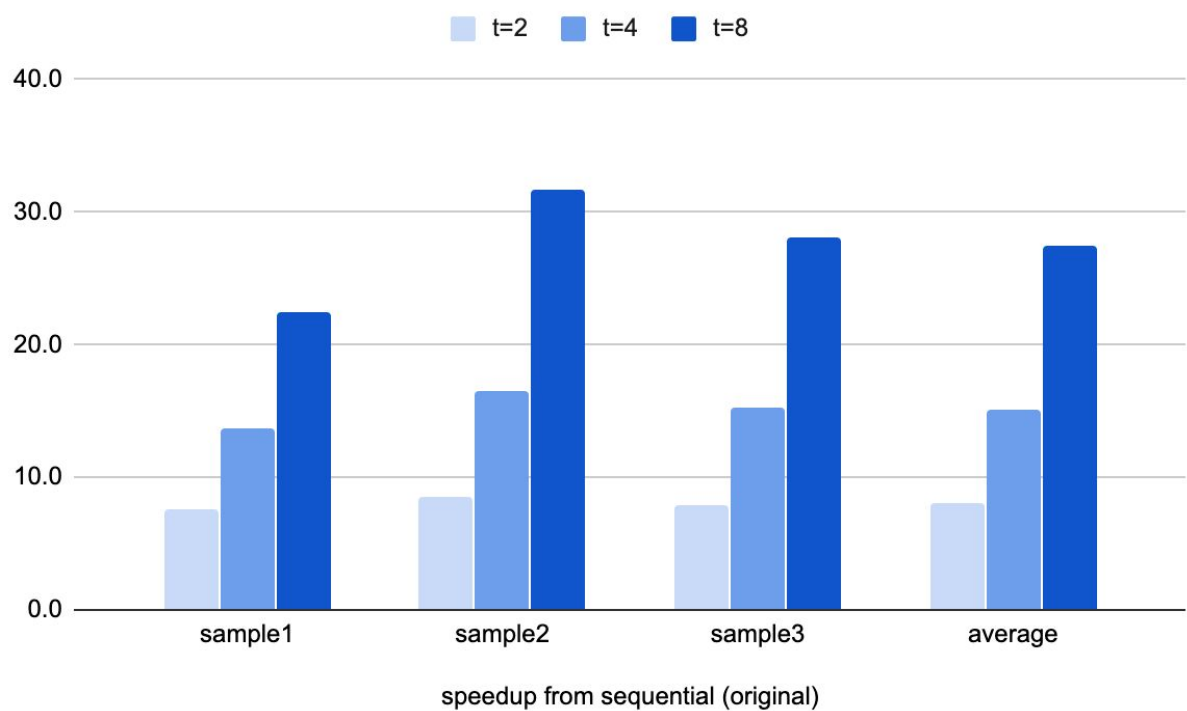
Note:

- By default, `t` is set to 1 so the program will run the improved sequential version.
- Run command `make clean` will clear the executable file as well as all generated output images.

Benchmark Results

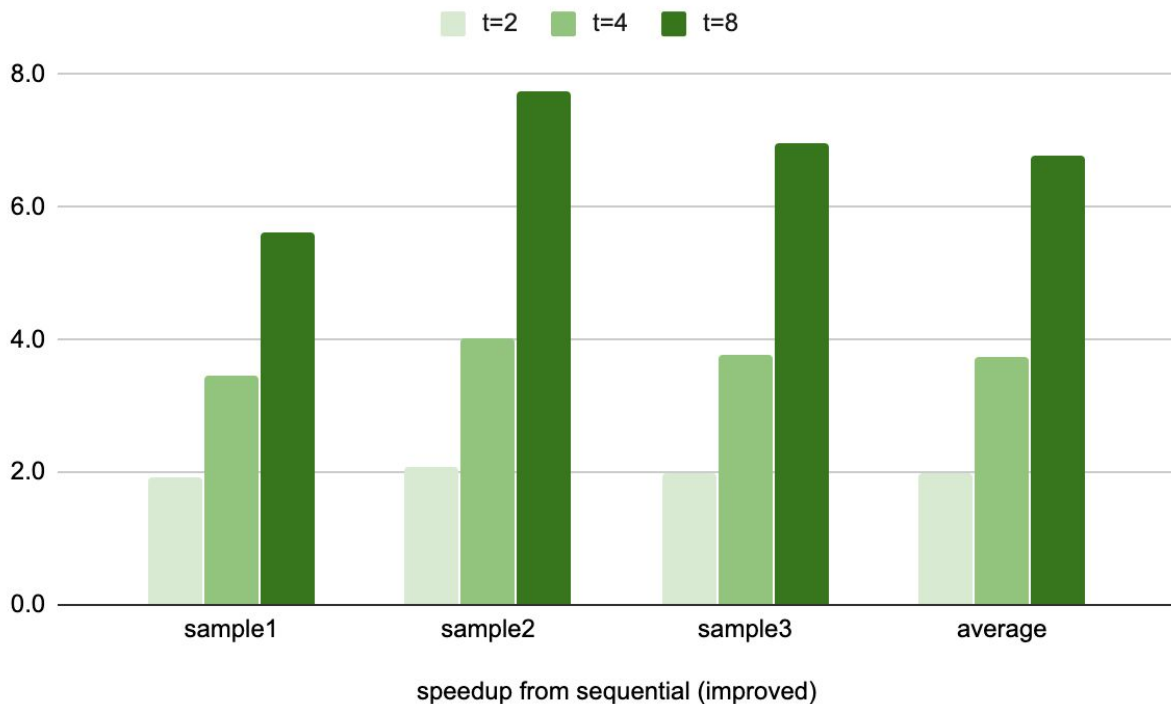
We benchmarked the parallel performance with running $t = \{2, 4, 8\}$ threads against two baseline sequential versions (see [Benchmarks](#) section above for details). **Below is an overview of the optimization results. Detailed data can be found in the [Appendix](#) section.** We observed that parallelism significantly improved the runtime of `compute_dist`, which is the most time-consuming section of the program and our target to optimize.

Compared with the **original sequential implementation**, we were able to achieve an average of 27x speedup across different samples when running the parallel program with 8 threads.



speedup from sequential (original)	t=2	t=4	t=8
sample1	7.6	13.8	22.4
sample2	8.5	16.5	31.8
sample3	8.0	15.2	28.1
average	8.04	15.17	27.45

Compared with our **improved sequential implementation**, we were able to achieve an average of 6.78x speedup across different samples when running the parallel program with 8 threads.



speedup from sequential (improved)	t=2	t=4	t=8
sample1	1.9	3.4	5.6
sample2	2.1	4.0	7.8
sample3	2.0	3.8	7.0
average	1.99	3.75	6.78

Limitation of speedup

The main reason for us not achieving ideal linear speedup is that there has to be a critical section in our implementation. During the computation of distance, we leveraged openmp parallel for loop to calculate distance for pairs of neighborhoods for each iteration. We also need to find the minimum of all the distances in order to create a threshold for possible neighborhood consideration in the next step. We are utilizing the separate accumulation technique which particularly uses a scratch vector and in the end do a reduction using the scratch vector. Finding the minimum in the scratch vector requires all threads to synchronize.

Since the minimum operation is commutative, we utilized the built-in OpenMP reduction operation and were able to minimize the cost of synchronization. However, multiple threads still need to synchronize in order to reduce, this part gives rise to limitation of speedup. All threads need to finish their own part of work in order to begin reduction. Therefore, uneven distribution of work (especially when the total amount of distance calculation can not be divided by thread count) would cause some threads waiting on others to finish their local computation. The total amount of work is determined by $X_{end} * Y_{end}$, which is the difference between input width/height and window plus one. For sample1, input dimension is 30 x 24, For sample2, input dimension is 93 x 92. For sample3, it is 49 x 48. This explains why sample 1 performs worse than the other two samples at 8 thread parallelization. At a window of 10, the total task number is $(30-1)*(24-1)=667$ which could not be divided by 8. Thus, it only achieved a 5.6x speedup compared to the improved sequential version while the other two samples that have task numbers that can be divided by 8 can achieve a 7x/7.8x speedup. We can see the same result works for 4 threads and 2 threads. Note that the divergence in performance is least obvious for 2 threads because the cost of one thread being idle is way less than 3 or 7 threads being idle.

Choice of machine

We are using the GHC cluster machine with the spec of 8 cores, each core is a I7-9700 cpu with 3Ghz and a local cache size of 12Mb. Compared to the lateday clusters where each core has 2.4Ghz and 15Mb L3 cache, we can see that although each core of the GHC machine has a smaller local cache, it has a better compute speed. This means that through proper assignment of resources to fit as much data into the local cache as possible, we can achieve better possible parallel performance.

Most image processing algorithms are run on GPU. The reason we are running on CPU is we think that our distribution of tasks is more coarse-grained. The reason for coarse-grained distribution of tasks is because we want to leverage temporal locality and avoid false-sharing by having a single thread take care of a collection of closely distanced neighborhoods since they share a large number of similar pixels. Since coarse-grained tasks are naturally suitable for CPU parallelization, we are using CPU instead of GPU.

References

Publications

- Efros, Alexei A., and Thomas K. Leung. "Texture synthesis by non-parametric sampling." Proceedings of the seventh IEEE international conference on computer vision. Vol. 2. IEEE, 1999.
<https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/papers/efros-iccv99.pdf>
- Wei, Li-Yi, and Marc Levoy. "Fast texture synthesis using tree-structured vector quantization." Proceedings of the 27th annual conference on Computer graphics and interactive techniques. 2000.
<https://graphics.stanford.edu/papers/texture-synthesis-sig00/texture.pdf>

Implementations

- <https://github.com/thuliu-yt16/TextureSynthesis>

Division of Work

Equal work was performed by both project members.

Appendix: All Benchmark Data

Measurement unit	ms
Machine Used	GHC machine (#46), with 8 cores
Core info	model: 158 model name: Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz cpu MHz: 799.987 cache size: 12288 KB cache_alignment: 64

Sample1 (sample1.ttr)

s1, w=10, r=100	original seq	improved seq	parallel t=2	parallel t=4	parallel t=8
startup	1	1	1	1	1
get_batch	0	0	1	1	1
compute_dist	10117	2342	1201	637	366
find_moves	9	9	12	13	13
unknown	1	1	2	2	2
elapsed	10128	2353	1217	655	385
s1, w=5, r=200	original seq	improved seq	parallel t=2	parallel t=4	parallel t=8
startup	1	4	4	4	4
get_batch	0	3	4	4	4
compute_dist	16943	4812	2475	1385	833
find_moves	71	70	85	91	102
unknown	4	5	10	11	14
elapsed	17019	4894	2578	1495	959
s1, w=20, r=100	original seq	improved seq	parallel t=2	parallel t=4	parallel t=8
startup	1	4	1	1	1
get_batch	0	0	1	1	1
compute_dist	6808	1631	851	460	279
find_moves	5	5	6	8	9
unknown	1	1	2	2	3
elapsed	6815	1641	861	472	293

Sample2 (1.ttr)

s2, w=10, r=100	original seq	improved seq	parallel t=2	parallel t=4	parallel t=8
startup	0	1	1	1	1
get_batch	0	0	1	1	1
compute_dist	226850	51659	25880	13271	6857
find_moves	153	150	158	166	173
unknown	1	2	3	2	2
elapsed	227004	51812	26043	13441	7034
s2, w=20, r=50	original seq	improved seq	parallel t=2	parallel t=4	parallel t=8
startup	0	1	1	1	1
get_batch	0	1	0	1	1
compute_dist	165757	45434	19863	10207	5233
find_moves	30	33	31	33	34
unknown	0	0	0	0	0
elapsed	165787	45469	19895	10242	5269
s2, w=80, r=100	original seq	improved seq	parallel t=2	parallel t=4	parallel t=8
startup	0	2	2	2	2
get_batch	0	1	1	1	1
compute_dist	362238	83394	42358	21893	11467
find_moves	10	11	12	10	12
unknown	1	3	3	2	2
elapsed	362249	83411	42376	21908	11484

Sample3 (2.ttr)

s3, w=10, r=100	original seq	improved seq	parallel t=2	parallel t=4	parallel t=8
startup	0	1	1	1	1
get_batch	0	0	1	1	1
compute_dist	50388	11560	5812	2995	1583
find_moves	39	35	37	41	47
unknown	1	1	2	1	1
elapsed	50428	11597	5853	3039	1633
s3, w=5, r=200	original seq	improved seq	parallel t=2	parallel t=4	parallel t=8
startup	1	4	4	4	4
get_batch	0	3	4	4	4
compute_dist	64674	18318	9117	4817	2612
find_moves	156	174	189	211	229
unknown	4	6	9	8	7
elapsed	64835	18505	9323	5044	2856
s3, w=20, r=50	original seq	improved seq	parallel t=2	parallel t=4	parallel t=8
startup	0	0	0	0	0
get_batch	0	0	0	0	0
compute_dist	26763	6362	3212	1647	861
find_moves	5	6	6	7	7
unknown	0	0	0	0	0
elapsed	26768	6368	3218	1654	868