Parallel Random-Art Hash Visualization: An Implementation with OpenMP

Yuxuan Zheng
New York University
yz10766@nyu.edu

Abstract

Random Art is a smart solution based on context-free grammar that makes hash comparison easier for human beings. In order to achieve pre-image resistance property, the expression tree formed by the grammar has to be deep and complicated, therefore rendering a random art image could be time consuming. In this paper, we look into two ways that we can parallelize the *Randon Art* algorithm. We find that on the pixel-rendering level, the algorithm is embarassingly parallel with respect to the number of threads. On the other hand, on the tree evaluation level, after we try different ways to optimize the algorithm, the performance gain is still negative.

1 Introduction

Hash visualization is a method that transforms cryptographic hash outputs, which are typically represented as alphanumeric strings, into structured images. This approach is based on the principle that humans are more efficient at identifying and comparing visual patterns than textual strings. A hash visualization algorithm maps the hash value to a deterministic visual representation, ensuring that small changes in the input produce noticeably different images. This method enhances usability in applications like root key validation and user authentication, where traditional textual comparison is prone to errors.

Random Art [1] is an implementation of hash visualization designed to convert a binary hash or other input strings into a structured and unique image output. The algorithm generates a mathematical function, structured as an expression tree, from a comtext-free grammar using a pseudo-random number generator seeded with the input. This function maps two-dimensional pixel (x, y) coordinates to color values in a fixed color space, typically RGB, in which case the mapping is $[-1, 1]^2 \to [-1, 1]^3$ (both the pixel cooradinates and the RGB are normalized). The generated image is deterministic, meaning the same input always produces the same output. Random Art ensures a wide variety of visual outcomes while maintaining sufficient regularity to be easily interpreted by humans.

Figure 1 are three example random-art images generated by the following grammar. The number served as the superscript is the probability that this branch is chosen during the expression tree construction.

$$
\begin{aligned}
& P \to (B, E, D) \\
& A \to X^{0.333} \mid Y^{0.333} \mid \mathrm{RAND}()^{0.333} \\
& B \to \mathrm{SIN}(D)^{0.5} \mid \mathrm{TAN}(B)^{0.25} \mid \mathrm{SQRT}(C)^{0.25} \\
& C \to A^{0,25} \mid \mathrm{ADD}(C, C)^{0.375} \mid \mathrm{MULT}(C, C)^{0.375} \\
& D \to \mathrm{ADD}(A, C)^{0.25} \mid \mathrm{MIX}(A, C, B)^{0.75} \\
& E \to \mathrm{ADD}(A, B)^{0.5} \mid \mathrm{MULT}(A, B)^{0.25} \mid C^{0.25}
\end{aligned}
\tag{1}
$$

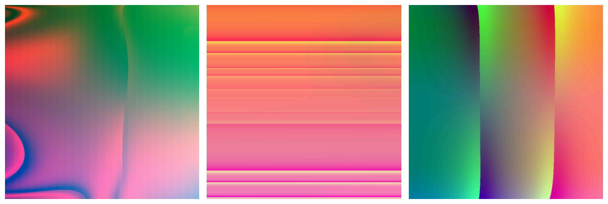
Figure 1: Sample random-art images generated by a simple grammar.

In order to achieve pre-image resistance property of a hash function, the expression tree should not be too shallow [1]. However, since the computation work grows exponentially with respect to the depth of the expression tree, the time taken for evaluation and generation of the images may be too long to afford. This problem is especially significant when we are generating a batch of images with a batch of seeds. To address this, we propose a *parallel random-art generation* algorithm that can parallelize the original algorithm on two different levels: parallel pixel generation or parallel expression tree evaluation. By leveraging multithreading using OpenMP, we hope there will be a speedup.

2 Literature Survey

We do an survey on how to parallelize the expression tree evaluation. The method on the OpenMP manual [2] is straightforward, which is to assign a `task` to each child during recursion. However, since the expression tree is not balanced in most of the time, doing so may lead to load-imbalance on each thread.

Another algorithm, called *parallel tree contraction* [3], proposes that a binary tree can be reduced by repeatedly performing *RAKE* and *COMPRESS* steps, where each step can be done in parallel. A generalization of the algorithm to trees of unbounded degrees is called *RAKE-SHUNT* algorithm [4], where the *SHUNT* operation is a *RAKE* followed immediately by a *COMPRESS*. This algorithm finishes in $O\left(\frac{N}{P} + \log N\right)$, where N is the number of nodes and P is the number of processors.

The above method seems to be a good way to parallelize the expression tree, but implementing it in C is very tricky. First, the original algorithm uses adjacency list to represent the tree. In our implementation, due to the nature of the randomly constructed tree, the tree is naturally represented as a pointer to the root node. This means that finding all the leaves takes constant time in the original representation, whereas it takes at leat $O(\log N)$ time in the pointer-to-root representation. Second, the *RAKE* and *COMPRESSION* steps are described as lambda calculus, which is natural to implement in a functional language. However, since C as an imperative language does not support partial functions, implementing such feature in C is very difficult.

3 Proposed Idea

The generation of a random-art image can be parallelized on two different levels:

1. Parallel pixel-rendering;
2. Parallel expression tree evaluation.

The pixel-rendering level can be parallelized easily. Since there are no dependencies between pixels and the work for rendering each pixel are the same, `static` scheduling should give the best performance because it has the least overhead.

The expression tree evaluation can be parallelized by assigning tasks to the recursive steps. How we assign the tasks and how many tasks we assign in total could be crucial to the overall speedup.

We can choose either of the above two levels of parallelism in an image-rendering task, but we cannot leverage both, because the two levels of parallelism use threads in different ways.

4 Experimental Setup

The experiment is done on NYU `crunchy1` machine. It has 64 (logical) CPUs. The CPU model name is AMD Opteron(TM) Processor 6272.

4.1 Parallel Pixel-Rendering

On this level, we use the grammar shown in Equation 1 to generate trees of depth 5, 8, 10, and 15 with 1, 2, 4, 8, 16, and 32 threads. Since the trees are randomly generated, for each depth we generate 10 trees. We render a 800×800 image based on the generated tree and calculate the average speedup.

4.2 Parallel Expression Tree Evaluation

We do experiments on three task assignment strategies:

1. Assign tasks in every recursion if `depth < THRESHOLD`;
2. Assign tasks only if `depth == THRESHOLD`;

3. Assign tasks only if `depth == THRESHOLD`, and in each recursion, instead of doing `depth++`, do `depth = (depth + 1) % (THRESHOLD + 1)`.

Then, again, we use the grammar shown in Equation 1 to generate trees of depth 5, 8, 10, and 15 with 1, 2, 4, 8, 16, and 32 threads.

After that, we change the grammar in which we use functions that takes more arguments, and repeat the steps above.

5 Experiments & Analysis

5.1 Parallel Pixel-Rendering

After generating trees of depth 5, 8, 10 and 15 with 2, 4, 8, 16, and 32 threads, we have the following plot of Speedup versus the number of threads shown in Figure 2.

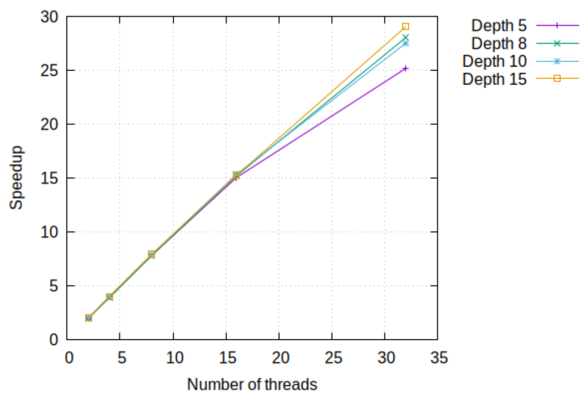

Figure 2: Speedup vs Number of threads

Figure 2 shows that the speedup is independent of the depth of the tree, and is almost equal to the number of threads used. This means that the task is *embarrassingly parallelizable* on the pixel rendering level.

5.2 Parallel Expression Tree Evaluation

The plots of Speedup vs the number of threads of the three strategies are shown in Figure 3, Figure 4, and Figure 5, respectively. We choose `THRESHOLD == 4` for creating a resonable number of tasks.

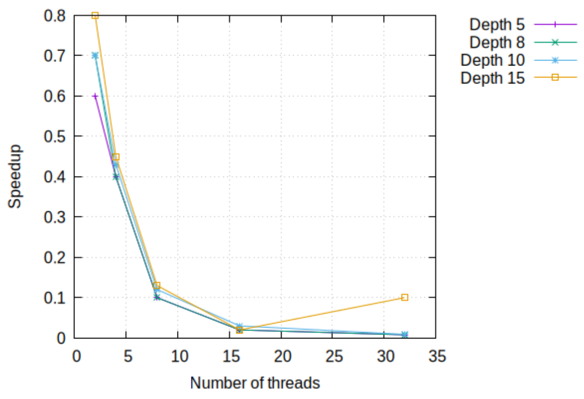

Figure 3: Strategy 1, THRESHOLD = 4

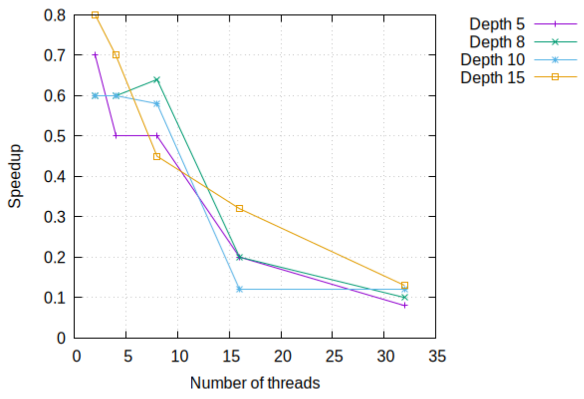

Figure 4: Strategy 2, THRESHOLD = 4

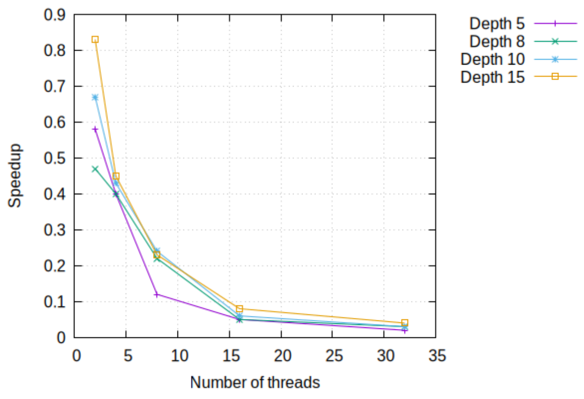

Figure 5: Strategy 3, THRESHOLD = 4

We can see that all speedup are less than 1, meaning that using multiple threads in this case slows down the image rendering. Also, the more threads we create, the worse performance it gets. The reason is that unlike parallel image rendering, where each thread does its own task without any synchronization, parallel expression tree evaluation has a `taskwait` to synchronize other threads before it can return. In addition, the overhead of creation of tasks cannot be ignored. All these reason lead to a significant drop in the performance.

On the otherhand, we change the grammar to see if the arity of functions influences the performance. We change to the grammar below:

$$\begin{aligned} &P \to (C, C, C) \\ &A \to X^{0.333} \mid Y^{0.333} \mid \mathrm{RAND}()^{0.333} \\ &B \to \mathrm{EIGHT\_SUM}(A, A, A, A, A, A, A, A)^{1} \\ &C \to \mathrm{EIGHT\_SUM}(A, A, D, E, A, B, D, E)^{1} \\ &D \to \mathrm{ADD}(B, A)^{0.25} \mid \mathrm{MIX}(C, D, E)^{0.75} \\ &E \to \mathrm{ADD}(A, A)^{0.5} \mid \mathrm{MULT}(B, B)^{0.25} \mid C^{0.25} \end{aligned} \tag{2}$$

Here, the EIGHT_SUM function takes 8 arguments (arity $= 8$), making the tree a "fat" one. By further modifying the rule, we can control the average arity in a given grammar. For example, the average arity of Equation 2 is $\frac{28}{10} = 2.8$. Table 1 is a comparison using 4 threads and depth equal to 15.

Average aity	**Speedup**
1.2	0.42
1.8	0.48
2.2	0.63
2.8	0.83
3.2	0.85

Table 1: Relationship between average arity and speedup

Although the speedup is still less than 1, we can see there is a performance gain as the average arity increases. This is because when the arity increases, a node has more children, and more tasks can be performed in parallel.

6 Conclusions

The random art image rendering is perfectly parallelizable on the pixel-redering level, whereas the gain is negative if parallelize on the tree evaluation level. This means that the task synchronization in a tree traversal is a bottleneck of the parallel algorithm. However, certain manipulations are useful to reduce the synchronization overhead, or to make it less significant compared to the time of completing a task, such as increasing the depth of the tree, or using a grammar with larger average arity (and hence larger average number of brunches). But overall, the "creating tasks inside recursion" method proposed in the OpenMP manual is not a good way of parallelizing the evaluation of expression trees.

Renferences

[1] A. Perrig and D. Song, "Hash visualization: A new technique to improve real-world security," in *International Workshop on Cryptographic Techniques and E-Commerce*, 1999.

[2] O. Board, "OpenMP Application Programming Interface Examples." 2022.

[3] G. L. Miller and J. H. Reif, "Parallel tree contraction and its application," in *FOCS*, 1985, pp. 478–489.

[4] A. Morihata and K. Matsuzaki, "A practical tree contraction algorithm for parallel skeletons on trees of unbounded degree," *Procedia Computer Science*, vol. 4, pp. 7–16, 2011.