Computers & Graphics 34 (2010) 585–593

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical Section

Parallel generation of multiple L-systems

Markus Lipp [a,*], Peter Wonka [b], Michael Wimmer [a]

[a] Vienna University of Technology, Austria
[b] Arizona State University, Austria

ARTICLE INFO

Keywords:
L-systems
Graphics hardware
Parallel processing
Real-time rendering

ABSTRACT

This paper introduces a solution to compute L-systems on parallel architectures like GPUs and multi-core CPUs. Our solution can split the derivation of the L-system as well as the interpretation and geometry generation into thousands of threads running in parallel. We introduce a highly parallel algorithm for L-system evaluation that works on arbitrary L-systems, including parametric productions, context sensitive productions, stochastic production selection, and productions with side effects. This algorithm is further extended to allow evaluation of multiple independent L-systems in parallel. In contrast to previous work, we directly interpret the productions defined in plain-text, without requiring any compilation or transformation step (e.g., into shaders). Our algorithm is efficient in the sense that it requires no explicit inter-thread communication or atomic operations, and is thus completely lock free.

1. Introduction

Procedural modeling techniques to compute large and detailed 3D models have become very popular in recent years. This leads to the question of how to handle the increasing memory requirements for such models. The current trend is towards data amplification directly on the GPU, for example tesselation of curved surfaces specified by a few control points. This results in low storage costs and allows generating the complex model only when needed (i.e., when it is visible), while also reducing memory transfer overheads. In the same vein, grammars can be viewed not only as a modeling tool, but also as a method for data amplification since a very short grammar description leads to a detailed model.

In this paper we investigate whether it is possible to efficiently evaluate one of the most classical procedural modeling primitives, L-systems, directly on parallel architectures, exemplified by current GPUs and multi-core CPUs. The main motivation is to enable interactive editing of large L-systems (examples are shown in Fig. 1) by designers, therefore it is important to speed up the computation of L-systems in order to achieve low response times.

Although L-systems are parallel rewriting systems, derivation through rewriting leads to very uneven workloads. Furthermore, the interpretation of an L-system is an inherently serial process. Thus, L-systems are not straightforwardly amenable to parallel implementation. Previous work therefore focused on specialized types of L-systems that do not allow side effects in productions, which makes them very similar to scene graphs [3]. In contrast, we deal directly with uneven workloads in L-system derivation, and we have identified two main sources of parallelism in the interpretation of L-systems: (1) the *associativity* of traversal in non-branching L-systems, and (2) the *branching structure* itself in branching L-systems.

The main contribution of this paper is a highly parallel algorithm for L-system evaluation that

- works on arbitrary L-systems, including parametric productions, context sensitive productions, stochastic production selection, and productions with side effects
- works directly on an input string and a plain-text representation of the productions without requiring any compilation or transformation step (e.g., into shaders)
- is efficient in the sense that it requires no explicit inter-thread communication or atomic operations, and is thus completely lock free
- parallelizes both within one L-system as well as among a large number of L-systems

To our knowledge, this is the first L-system algorithm that is highly parallel, i.e. utilizes thousands of threads in an efficient manner. This is achieved by identifying and exploiting the parallelism inherent in L-system derivation using parallel programming primitives like scanning or work-queue management, and a novel algorithm to explicitly resolve the branching structure. We demonstrate that our algorithm outperforms a well optimized single-core CPU implementation on larger L-systems.

This paper is an extended version of [5], adding support for multiple L-systems as described in Section 6.

* Corresponding author.
E-mail addresses: lipp@cg.tuwien.ac.at (M. Lipp), pwonka@gmail.com (P. Wonka), wimmer@cg.tuwien.ac.at (M. Wimmer).

0097-8493/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2010.05.014

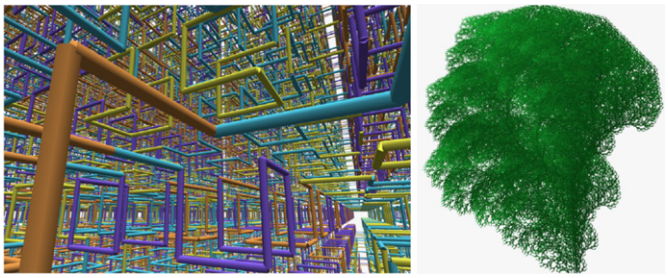

Fig. 1. L-systems generated in real-time, at up to 198,000 modules per millisecond: Hilbert 3D space-filling curve and 2D plant.

Overview: First we will provide a background on L-systems and parallel primitives in Section 2. An analysis of the intrinsic parallelism of L-systems is provided in Section 3. Then our system consisting of two major building blocks will be described: (1) The derivation step will start with the axiom and generate a long string of modules (Section 4). (2) The interpretation step takes the string as input and generates the actual geometry (Section 5). An extension to support multiple independent L-systems in parallel is shown in Section 6.

1.1. Previous work

General L-systems: Prusinkiewicz and Lindenmayer cover the basic L-system algorithm [9]. Multiple extension to the basic approach were introduced [10,11,7].

Parallelizing L-systems: Lacz and Hart showed how to use manually written vertex and pixel shaders combined with a render-to-texture loop to compute L-systems [3]. This concept was later extended using automatically generated geometry shaders [6]. Both methods require a shader compilation step for the productions. Further a transformation step of every production's successor to a set of successors is needed to allow independent parallel executions in a shader. For example, the production $L \rightarrow aLf[+L]Lf[-L]L$ is transformed to the set $L \rightarrow aL, af+L, afL, aff-L, aff-L, affL$ [3]. This is only valid if the successor of L does not have any effect on the traversal state, which is not generally the case.

An algorithm utilizing multiple processors (the results show up to eight CPUs) with distributed memory, communicating using the Message Passing Interface (MPI) was introduced [14]. In their algorithm, the derivation of the L-system is performed using two binary trees, a Growth-State Tree (GST) and a Growth-Manner Tree (GMT). To actually render the system, the GST is interpreted as a scene graph. In order to get global scene-graph transformation matrices needed for rendering in the individual threads, the matrices are serially transferred from one process to the next.

Parallel computation in CUDA: In order to access the parallel computing capabilities of GPUs we employ the NVIDIA CUDA data-parallel programming framework [2]. Recent work shows how to map computations having a highly dynamic nature to CUDA. Most notably, algorithms to efficiently implement work-load balancing using a compactation step were introduced in the context of KD-trees [15], Reyes-style subdivision [8] and bounding volume hierarchies construction [4]. Generalized stream compaction was presented by Billeter et al. [1]. In the context of tessellating parametric surfaces, scan operations were used in order to scatter dynamically generated vertices to a VBO [12]. We employ both work-load balancing and vertex scattering in our work.

2. Background

Our work is based on L-systems and parallel processing primitives. Both concepts will be explained in this section.

L-systems. In our work, we use the formalism of parametric L-systems as introduced by Prusinkiewicz and Lindenmayer [9]. Parametric L-systems operate on *parametric words*, which are strings of *modules* consisting of *letters* with associated *actual parameters*. An L-system consists of a parametric word ω called the *axiom*, and a set of productions describing how the current word is transformed. A production consists of a letter possibly combined with *formal parameters*, called the *predecessor* and a *successor*. The successor consists of a list of letters, where each letter can have multiple *arithmetic expressions* containing formal parameters. Formal parameters can be global or local to one production rule. The real-valued *actual parameters* appearing in the words are calculated from the arithmetic expressions of *formal parameters*. The predecessor can also consist of several letters, in which case the L-system is called context sensitive [9].

In the following example, F, A, and B are the letters defining modules, g_i are global parameters, l is a local parameter, and the arrow separates the predecessor from successor:

$$F(l) \rightarrow A(l^* g_1)[B(l+g_2)]$$

To actually generate geometry, two distinct phases are performed: A *derivation* phase generating a string of modules, and an *interpretation* phase in which the string of modules is interpreted in order to generate geometry.

Derivation: The derivation starts from the axiom. For every module contained in the axiom, a *matching* production is searched. A production matches a module m if the letter of the predecessor matches the module letter, and the number of actual parameters in the module equals the number of formal parameters in the production. We then *apply* the matching production to the module: First, for every module in the successor, we calculate the actual real-valued parameters from the arithmetic expression of the formal parameters. Then we *rewrite* the module m with the modules of the successor. One *iteration* consists in rewriting all modules in the string *in parallel* using matching productions [9]. A user-defined amount of iterations is performed in order to get the final string of modules.

Interpretation: The interpretation is performed *serially* from the start of the string, performing modifications of a *turtle state* based on predefined *turtle commands* associated with specific letters [9]. The turtle state represents the position and orientation of a virtual turtle. This state can be represented with a 4×4 matrix. The turtle commands associated to letters modify the turtle state, for example 'F' moves the turtle forward while drawing a line, or '+' rotates the turtle. Most of these turtle commands can also be expressed by a 4×4 matrix. A notable exception are the commands '[' and ']', which push and pop the turtle state on a stack, allowing the creation of *branching* (also called *bracketed*) L-systems [9].

Parallel primitives. We extensively use the parallel *scan* primitive in our work. Given an ordered set of values $[a_0, a_1, \ldots, a_n]$ and an associative operator $\circ$ with the identity element I, an exclusive scan operation will result in the ordered set $[I, a_0, a_0 \circ a_1, \ldots, a_0 \circ a_1 \circ \ldots \circ a_{n-1}]$ [13]. If the operator is the addition, this results in a set of values s_i with $s_i = \sum_{j=0}^{i-1} a_j$. The main advantage of the scan primitive is its capability to compute seemingly serial operations very efficiently on highly parallel hardware, since subsequences can be processed independently due to associativity. Unless noted otherwise, we always refer to an exclusive scan on integral values using the addition operator when we use the term scan in our work.