Distributed Context-Aware Visualization

Harald Sanftmann*, Nazario Cipriani†, Daniel Weiskopf*

*VISUS, Email: {sanftmann—weiskopf}@vis.uni-stuttgart.de
†IPVS, Email: Nazario.Cipriani@ipvs.uni-stuttgart.de
Universität Stuttgart, Stuttgart, Germany

Abstract— We present a visualization framework integrated in a context-aware system that uses a common underlying stream processing middleware for tight integration of data accessing, processing, and visualization. Context-aware systems are often realized on mobile devices that do not have the computational power to perform complex tasks. Therefore, a dedicated hardware infrastructure might be required for data processing. In our case, stream processing is used, supporting parallelism on distributed and shared memory multiprocessors. We present the integration of visualization modules into a Java-based stream processing framework for context-aware systems, with focus on efficient communication and parallelization. Our approach is demonstrated for the example of a flow visualization scenario.

Keywords—Distributed/network graphics; visualization; context-aware;

I. INTRODUCTION

We present our visualization system which is embedded in a context-aware system and uses the hardware environment targeted and the computing infrastructure provided. Context-aware systems [11], [22] have been emerging with the trend toward ubiquitous and mobile computing. They use context information (especially the user’s current positions and situation) to react to changes of the environment. Context information can be acquired and used by individual applications separately or managed by a dedicated platform for context data. Such platforms can collect data from different data providers and provide an interface to query that data. Typical end user devices are mobile devices such as mobile phones, smartphones, PDAs, or laptop computers. Since mobile devices or netbooks are less powerful than typical desktop PCs or workstations, additional hardware infrastructure is required for data processing and rendering. Moreover, context data changes over time and thus continuous data processing is needed.

Stream processing systems have been developed in the database community which are able to answer such continuous queries by mapping them to a network of operators managed and executed by a processing middleware. The NexusDS system by Cipriani et al. [10], [17] was the first stream processing system from the database community targeting context data processing and considering visualization as promising application. NexusDS is part of the Nexus [12] platform, which provides a flexible middleware for context-aware applications. Nexus is designed as an open platform where everybody can contribute context information that is federated within the platform. NexusDS supports structured and unstructured data, which is necessary to produce useful image output. NexusDS uses the JXTA peer-to-peer protocol to enable communication over different network protocols on different architectures supporting TCP/IP and Bluetooth communication, discovery, and NAT-traversal. NexusDS operators are executed only when new input data arrives and new output needs to be generated supporting power-aware computation. These properties make NexusDS well suited for a wide range of pervasive computing applications, for example integration of geo-spatial simulation results such as wind flow, contamination, or disaster recovery.

In this work, we extend the Java-based NexusDS framework by allowing the development and integration of C/C++ modules, which are most commonly used in the visualization community. We rely on quite common modules from the visualization point of view, but perform a tight integration in the operator concept. NexusDS supports parallelism on distributed and shared memory multiprocessors. Therefore, we focus on efficient communication and extend the framework by a technique which helps to easily parallelize operators exploiting data parallelism.

The computational resources needed for a visualization may not be constant: they can depend on the input data (different data may require more computational work), the visualization accuracy needs (some applications may require higher accuracy than others), or the targeted level of interactivity (some applications like for augmented reality require low latency, while a delay of a few seconds may be acceptable for WEB-based monitoring applications). Through stream processing we obtain a highly adaptive and scalable solution: the same visualization pipeline structure can be used to fulfill all these requirements. For example, in case of low hardware requirements, the visualization pipeline can be deployed to a single PC running multiple visualizations. In case of higher resource needs, multiple PCs can be used to execute the same visualization pipeline, leading to low latencies and high data and image throughput.

Given the large variety of data sources provided by a context-aware platform various new applications can be realized with our system which benefit from scalable visualization capabilities.
II. RELATED WORK

This section reviews the main fields of previous research related to our paper: context-aware systems and parallel visualization. Both fields are traditionally covered by mostly disjoint research communities. However, the trend toward ubiquitous information systems with heterogeneous data sources has recently brought these fields closer together. In particular, the massive increase in available data from sensors and computer networks in context-aware systems has emphasized the need for appropriate visual data analysis and presentation.

A typical example of a platform for context-aware mobile applications is given by Raento et al. [21], where information of mobile phones such as position, phone profile, and last phone usage are collected and made available to the persons in the contact list. An alternative operator-based context-aggregation platform was presented by Chen and Kotz [9]. PLACE* [29] is a distributed spatiotemporal data stream management system for moving objects. PLACE* supports continuous spatiotemporal queries that are evaluated by a network of regional servers. A query is continuously answered by a querying server, a tracking server, and a set of additional participating servers. While any of these context-aware systems supports accessing context data in some way or another, they do not facilitate efficient, low-latency, and generic visualization techniques.

Augmented reality systems also provide narrow context information like position and pose estimates of mobile objects, but offer only a limited scalability. Shibata et al. [24] present a scalable architecture, however scalability is limited to the ability to serve clients with different capabilities by one server, without considering parallelization of the rendering or visualization on the server side.

Computer graphics and visualization considered efficiency issues early on. Parallelization in particular has been a popular means of increasing processing speed. Early work on parallelization in the area of visualization concentrated on parallel rendering [19]. In contrast, we consider parallelization of the visualization process as a whole and not just its rendering stage. A good description of the generic concepts of task, data, and pipeline parallelism in visualization is presented by Ahrens et al. [4]. The data-flow paradigm is the common basis for typical visualization processing, as used in scientific visualization environments like AVS [25], SCIRun [18], VTK [23], and COVISE [1]. The data-flow approach can be combined with parallelization to support fast processing of large data sets. However, beyond the common data-flow model and parallelization strategies, visualization systems differ in the details of data communication and workload distribution.

AVS [25] and COVISE [1] use a demand-driven execution model with a centralized executive. SCIRun [18] as well has a centralized executive. The Visualization Toolkit [23] (VTK) is an open source visualization toolkit with a demand-driven update semantics. Ahrens et al. [3], [4] saw that designing an efficient mechanism for controlling many processes from one centralized executive is difficult and developed a parallel extension to VTK that has no centralized executive. In another extension for VTK, Moreland et al. [20] included parallel rendering components. Finally, Dutra et al. use VTK for distributed visualization [13]. Similar to our work, they use a Java-based toolkit to allocate resources and to communicate data. They propose a master-slave architecture where the master splits the data and merges the final results. This differs from our approach where the deployment of operators is more flexible. ParaView [2], [8] uses VTK as data processing and rendering engine and avoids the use of a centralized executive to obtain a more scalable solution than AVS Express or SCIRun. In contrary to our system, it has a demand driven update semantics and only focuses on exploiting data parallelism but not task and pipeline parallelism.


NMM by Lohse et al. [16] is a stream processing framework running on distributed and shared memory multiprocessors targeting multimedia applications. In contrast to our framework it has no visualization modules. FlowVR [5] is a middleware for distributed and shared memory parallel systems targeting virtual reality and scientific visualization applications dealing with large displays. On the one hand FlowVR offers an abstraction from the distribution of modules over a network, but on the other hand it encourages the usage of MPI for high performance applications which seems unreasonable. In contrast to our system it does not focus on context data processing.

NexusDS [10] is a flexible and scalable middleware that is highly customizable. The complete system is implemented using the Java programming environment. While different elements of NexusDS appear in the above prior systems at various places, NexusDS is unique in its combination of event-driven communication (simplifying parallelization), decentralized execution (better scalability), and support for distributed and shared memory systems. This new combination of methods allows us to integrate parallel processing and visualization with high throughput and low latency in general streaming environments.

III. CONCEPTUAL MODEL AND STRUCTURE

A. NexusDS Operator and Streaming Model

Now we give a brief overview of the functionality of NexusDS which is essential in order to understand our extensions to NexusDS and some configuration choices we made.
The distributed stream processing framework consists of a set of stream processing nodes that are capable of finding each other in the network, accepting processing requests, and delivering results. A processing request is represented by a set of operators that form a processing graph. Operators have several input and output ports that are interconnected, thus forming a network of operators. An execution environment is available to execute the operators.

NexusDS allows parallelization on distributed and shared memory systems. Modules running on the same machine are executed in one process, the common address space allows sharing data efficiently. To communicate data between modules running on different machines, data is serialized, sent over the network, and deserialized by the receiver.

The stream processing system is suitable to handle time varying data. To hide latencies introduced by the network transfer, processing and receiving of data is asynchronous. Each input port has a separate input queue. When data arrives it is processed; the results are pushed to the input queues of all operators connected to its output port. Therefore no centralized execution module or scheduler is needed to drive the stream processing system.

NexusDS provides abstract operators, so-called platform sinks and sources, which can handle sending and receiving data over the network between fragments during runtime. These operators allow to communicate data over the network asynchronously.

### B. Graphical Editor

After the development of the operators they have to be interconnected. We developed a graphical editor, which is similar to other dataflow-based visual programming environments. Available operators are stored in a repository which is read by the editor. Operators can be dragged and dropped to a GUI drawing area where the interconnections between the operators can be defined.

Once the visualization pipeline is designed it can be separated, within the graphical editor, into one or more so-called fragments. Each fragment groups operators that should run on the same stream processing node. During this step necessary platform sinks and sources are created automatically.

### C. Deployment and Execution

After design time each visualization pipeline fragment needs to be assigned to a stream processing node. Several fragments may be assigned to a single node. The assignment is specified by the user taking into account the computational capabilities of the nodes and the interconnection between them.

Once all mapping information is available, the visualization pipeline fragments are deployed on the corresponding stream processing node. Each node is responsible for establishing the proper connection to its neighbor and for receiving, processing, and sending data in a push-based manner.

As a final result, the output of the processing pipeline can be displayed on the stream processing node running the rendering operator, or images can be sent to a image client operator for display on mobile devices.

Distributed-memory and shared-memory task decomposition and pipelining are supported by the NexusDS framework by placing operators onto different computational nodes or to one node respectively.

### D. Domain Decomposition

We briefly sketch how we realized domain decomposition with NexusDS. Fig. 1 illustrates domain decomposition for an example where an operator is decomposed into two independently working operators. The number of array elements may change for each data packet. Moreover, the number of operators can vary, as specified by the module parameters during module initialization.

### IV. DATA TRANSFER AND EXECUTION MODEL

#### A. Efficient Communication

NexusDS is implemented in Java. For the development of efficient visualization operators we chose the C++ language. The Java Native Interface (JNI) is used to call C/C++ code from Java. To implement efficient communication between the C++ visualization operators over the Java NexusDS framework we decided to use java.nio.ByteBuffer objects. These objects are wrappers for C++ memory blocks, so that C++ objects can be handed over to Java without copying the data itself. The Java NexusDS system can transport references to data between operators if the operators are on the same node, see Fig. 2. We made a design decision not to use linked but flat data structures. If the operators run on different nodes, flat data structures also prove beneficial. Since the data is already in a linear memory...
block, it can be sent over the network and stored in a local memory block on the receiver side. For this task we extended NexusDS with platform sinks and sources which can handle java.nio.ByteBuffer objects.

**B. Memory Management**

The operators allocate memory for their outputs on the native (C++) side. They just hand over constant references to other operators, i.e., operators are not allowed to modify input data, but they can output more than one reference to the same memory to different operators. The communication channels are unidirectional and thus there is no mechanism to explicitly tell an operator that a given input is not needed any more. Therefore, we have implemented a different mechanism: the input becomes invalid after the next data packet has been received on the same input port. Analogously, the memory of an operator output has to be kept until another output element is sent via the same port. This implies that the receiving operator should block if data is coming through an input port when the previous element that came through the same input port is still needed for computations. In contrary to the solution presented by Ahrens et al. [4], local copies are entirely avoided using this paradigm. As NexusDS uses internal queues for the input ports, the number of outputs an operator has to keep increases with the size of these queues.

We configured NexusDS to use FIFO queues in our system. The size of the input queues needs to be small to keep latencies low. In case of uneven concentration of incoming events the queues should be increased to allow high utilization of nodes. (This is not an issue in the example scenario of Section V.)

**C. Thread Model**

NexusDS is responsible for the data transfer between operators and also provides an execution model. An operator can have several input ports, see Fig. 3. Each port has an input queue for incoming data. We configured NexusDS to associate one thread to each input queue, this thread is waiting until the input queue is not empty. Then the respective input method of the Java operator is called, which calls the native input method of the C++ operator. The operator processes the data and eventually outputs results to some output ports. Since each input queue delivers input data to the operator concurrently, synchronization has to be performed by the C++ operator.

**D. Communication over the JNI**

For the communication between the Java and the C++ side we made use of the Call-Invoke pattern [27]. To output data from the C++ operators each operator has one registered listener that sends the output of all output ports back to the Java part of the operators.

1) **Resolving a C++ Object over JNI:** It is not possible to call a C++ object method over the JNI, but only C functions. In case of multiple operator instantiations, it is therefore necessary to call the right instantiation. This problem is solved by storing a C-pointer inside the Java classes that is passed to the C function as an additional argument, which is then used to resolve the right C++ object.

2) **Multi-threaded Execution:** Since a separate thread is associated with each input port, the operators receive data in different execution threads. To resolve the Java object and the method for sending back the operator output, a JNI interface pointer and the jobject pointer are needed. These are valid only for the thread associated with it [15]. By saving these pointers in thread local storage we are able to resolve the right pointers and allow input queue threads to run concurrently on the C++ side.

**V. CONTEXT-AWARE MOBILE VISUALIZATION**

Our framework can be used to facilitate ubiquitous computing by implementing demanding visualizations based on context data for mobile clients.
In our example, air flow in a room is simulated and visualized (see Fig. 4). However, due to the generality of NexusDS and the tight integration of visualization modules, any other visualization example should be workable within our environment. For general background information on flow visualization, we refer to overview presentations like [26].

The operators for the whole data generation and visualization process are shown in Fig. 5. Dynamic changes in the scene are input by tracking operators. The status of the windows and doors are obtained by computer vision techniques and are already stored in a database, called the context server. Data which does not change too frequently and is not time-critical can be stored in the context server. The Window-Tracker module streams this data in by regularly querying the database and generating output when the status changes. The MobObj-Tracker module is responsible for tracking moving object in the room, which is an example of a permanently updated context-data source. Eissele et al. [14] considered the tracking quality in a similar scenario as additional context information.

The changing geometry is voxelized and sent to a solver operator that calculates the velocity in the simulation volume (using a finite-volume numerical solver for computational fluid dynamics). The output of the solver operator is the air flow velocity field discretized on a regular grid. Then, the velocity field is sent to interpolation and velocity magnitude calculation operators. Their results are used to automatically place seed points for streamlines at appropriate positions to capture the important features of the flow. The most time-consuming task is the subsequent computation of streamlines (CalcStr) by solving the ordinary differential equation for particle tracing (with 4th-order Runge-Kutta integration and tricubic interpolation). Based on the streamlines stream ribbons are calculated (CalcRib), which are visualized by one or more render clients. Render operators can also output an image stream which can be sent to remote targets (e.g., mobile graphics devices) by image client operators. Constant generation of new data is triggered by the tracking and solver operators, leading to data streaming through the complete process of Fig. 5.

The islands in Fig. 5 represent different computational nodes. The links between the operators are just partially shown to avoid clutter in the diagram. The render operator, for example, can also display streamlines or LIC (line integral convolution) and needs the tracked positions of mobile objects connected to the input ports, which are left empty in Fig. 5.

The render operator uses the OpenSceneGraph (OSG) application programming interface. The test platform consisted of PCs with Core 2 Quad Q6600 CPUs, 4 GB DDR2 main memory, and gigabit Ethernet interconnection.

We ran different configurations to validate the efficiency of our framework (see Fig. 6). For the measurements we neglect the tracking operators, which are asynchronously feeding new positions of the tracked objects to the solver operator, and we consider a single render operator. We started with a single-threaded version, with one single...
streamline calculation operator (VC), where the operators run on one node and were linked directly by listeners, without the overhead of the communication over the JNI. Next we created a single-threaded configuration where the operators were instantiated in Java and were connected in the Java classes (VJ). So we could measure the overhead of the communication over the JNI. Then we made use of NexusDS to run the operator-network on a single node (F1). As the framework creates one thread for each input port of an operator this was a multi-threaded version. In this way the Interpolation operator and the Magnitude operator were running concurrently, exploiting task parallelism. With this configuration the overhead of our framework is tested. The measurements show that approximately 95% of the time is spent in the streamline calculation (see Fig. 6 VC).

To show the scalability of our framework, we created several configurations where we parallelized the streamline calculation with the domain decomposition presented in Section III-D. We created configurations with 2 and 4 CalcStreamline operators running in parallel (F2, F4) on one node together with all other operators. Since OSG, which was used for final rendering, consumes a full CPU core we moved the rendering and the stream ribbon calculation parts to a separate computing node (F4S). Finally, we created configurations with 8 and 16 parallel streamline calculation operators, with up to 4 streamline calculation operators running on one node and with the stream ribbon and rendering operators running on a separate node (F8S, F16S).

From the measurements we can conclude that there is no significant overhead – neither for the thin Java communication interface nor for the streaming operator framework. By moving the stream ribbon calculation and the rendering to a separate node, we obtain an improvement regarding the streamline calculation time (see Fig. 6 F4 and F4S). Since the results of the streamline calculation now have to be transferred to a different node, the transfer time is growing but the configuration pays off since all four cores are available for streamline computation. The setup is even more beneficial when several streamline calculations need to be performed and pipeline parallelism is exploited.

Our framework focuses on providing scalable parallelized visualization capabilities for context-aware mobile systems which could be used in ubiquitous graphical information systems especially in geo-information context.

VI. CONCLUSIONS AND FUTURE WORK

We have shown how to integrate common visualization modules in a stream processing framework dealing with context data. We put special attention to efficient communication and developed a scalable adaption mechanism for data parallel processing. In addition to parallelized visualization, our framework is designed to transparently fit into large, heterogeneous systems – both in terms of hardware infrastructure and software architecture. Therefore, stream processing for visualization has been included in a flexible and scalable middleware (NexusDS) for heterogeneous and open information systems. In particular, context-aware systems are supported – as the basis for ubiquitous visualization and graphics. We have shown how visualization computations can be formulated in the form of operators that communicate through the middleware. Special attention has been paid to issues of memory management and programming language interoperability (C/C++ for typical visualization implementations vs. Java for typical middleware and communication software). Despite the flexibility of the middleware-based approach, we still achieve reasonable speed-up for parallel execution on small to medium-sized systems (up to 16 parallel operators in our tests).

In future work, we would like to extend our approach by including automatic assignment of resources [28] and dynamic adaptation. Domain decomposition for the parallelized operators could be modified at runtime, reacting to possible changes introduced by the environment, e.g., available server hardware infrastructure, varying number of users and processing requests, changing amount of visualization data, interconnect, or reacting when a node comes down.

REFERENCES


