EGPGV '06

Eurographics Symposium on Parallel Graphics and Visualization '06

Symposium Short Papers Proceedings

A. Heirich, B. Raffin, and
L. P. dos Santos (eds.)

May 11-12, 2006
Braga | Portugal

EGPGV 2006

6th Eurographics Symposium on Parallel Graphics and Visualization

Braga, Portugal
May 11 th – 12 th, 2006

Short Papers Proceedings

Symposium Chair
Luis Paulo Santos, Universidade do Minho, Portugal

Programme Co-Chairs
Bruno Raffin, Laboratoire Informatique et Distribution, INRIA, France
Alan Heirich, Sony Computer Entertainment America, USA

Sponsored by EUROGRAPHICS Association in cooperation with ACM SIGGRAPH

Published by Departamento de Informática, Universidade do Minho, Braga
ISBN: 978-972-99166-3-2

Preface

This year the symposium received 30 submissions. These were reviewed by the 23 member program committee and by the editors. Most papers received three reviews, although a few received two. Every paper was also read by one or both editors. In the end the reviewer rankings correlated well with the editors feelings about the papers and the selection process was easy. The final program contains the 19 best submissions (as judged by us) for an acceptance rate of 2/3.

As in recent years, this year most of the papers concerned scientific visualization of large data sets. Two sessions and the second keynote were devoted entirely to commodity visualization clusters, and many other papers generated results using clusters. Other sessions were on volume rendering, large data and visualization, and parallel rendering and animation.

The first keynote, *Rendering on Demand* by Alan Chalmers of Bristol University, was on parallel photorealism and reminded us of the origins of this symposium series.

The first session on "Parallel Rendering and Animation" demonstrated the range of topics in the symposium. *An application of scalable massive model interaction using shared-memory systems* demonstrates interactive ray tracing on a complete Boeing 747 airplane model. *Accelerating the irradiance cache through parallel component-based rendering* uses task and data parallelism to speed up indirect illumination calculations in a physically based ray tracer. *Parallel cloth simulation on distributed memory architectures* addresses collision detection in a distributed memory environment where the data set must be partitioning across processors. *Dynamic load balancing for parallel volume rendering* proposes a k–d tree algorithm for rapidly rebalancing level-of- detail rendering techniques.

The second session was dedicated to parallel volume rendering. *Interactive volume rendering of unstructured grids with time-varying scalar fields* addresses problems of compression and prefetching unstructured data, and rendering it on the GPU using a dynamic level-of-detail technique. *Optimized volume raycasting for graphics-hardware-based cluster systems* uses a combination of techniques including k–d tree load balancing and empty space skipping for interactive volume rendering on a cluster. *Accelerated volume rendering with homogeneous region encoding using EACD on GPU* presents an improved space skipping algorithm for volume rendering using 3D texture mapping.

The third session of day one was on the general subject of Visualization. *Parallel texture-based vector field visualization on curved surfaces using GPU cluster computers* presents a hybrid of image-space and object-space, sort-first and sort-last techniques. *Distributed force-directed graph layout and visualization* parallelizes the Fruchterman- Reingold layout algorithm on a cluster. *Time step prioritising in parallel feature extraction on unsteady simulation data* explores methods to reduce the interactive response time when visualizing large CFD data sets. *Parallelization of inverse design of luminaire reflectors* presents an application of shape optimization in lighting design that is solved on a cluster.

The second day opened with a keynote on cluster rendering, *The Challenges of Commodity Visualization Clusters* by James Klosowski of IBM. This was followed by two sessions on clusters.

WinSGL: software genlocking for cost-effective VR installations under Microsoft Windows provides a solution under Windows to a critical problem in constructing tiled high resolution displays. *Sorted pipeline image composition* demonstrates a practical technique that produced significant speedup for sort-last architectures.

Optimized visualization for tiled displays implements a multicast protocol to reduce the overhead of tile-sort in Chromium sort-first applications. *Parallel particle rendering: a performance comparison between Chromium and Aura* demonstrate the advantages of using scene-graph level parallel decomposition. *Piggybacking for more efficient parallel out-of-core isosurfacing* presents a technique to eliminate redundant communication in a Marching Cubes algorithm on a cluster.

The third session of day two was on large data visualization. *A scalable, hybrid scheme for volume rendering massive data sets* combines object-space and image-space parallelism and is demonstrated on 400 processors of a cluster. *Remote large data visualization in the ParaView framework* presents a client-server framework for remote visualization implemented in the popular *ParaView* visualization system. And *Multi-layered image caching for distributed rendering of large multiresolution datasets* demonstrates an image caching optimization for progressive refinement rendering of a terascale dataset.

The symposium ended with a series of six short papers. *Pipelined sort-last rendering: scalability, performance and beyond* presents a performance analysis of pipelined sort-last rendering for polygonal and volume rendering. *A simple, distributed rendering toolkit* demonstrates a rendering system for clusters driving multi-screen Virtual Reality hardware. *Hardware for a ray tracing technique using plane-sphere intersections* proposes a hardware system that parallelises radiated from the same point, using plane-sphere intersections. *Accurate workload estimation for fast parallel RMT* evaluates a model that estimates isosurface extraction efforts for the Regularised Marching Tetrahedra. *A distributed memory GPU implementation of the Boris particle pusher algorithm* explains the implementation of this algorithm on stream processors, particularly, GPUs with programmable shading capabilities. *3D critical points computed by nested OpenMP* presents a parallel approach to find critical points, essential for velocity field topologies.

This collection of papers shows that parallelism is alive and well in scientific visualization and rendering and that clusters are the preferred hardware environment. While many of the paper authors are familiar faces there were also many first-time contributors. The paper authors, like the program committee, were evenly balanced between the USA and the international community, primarily European.

Alan, Bruno, and Luis
Braga, Portugal, May 2006

Sponsors

Universidade do Minho

FUNDAÇÃO CALOUSTE GULBENKIAN

CCTC
CENTRO DE CIÊNCIAS E TECNOLOGIAS DA COMPUTAÇÃO

BRITISH COUNCIL

camel active
REALITY WEAR

Centro de Computação Gráfica

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

Cover Image Credits

front cover:

The model is courtesy of Veronica Sundstedt, Bristol Graphics Group, University of Bristol.
K. Debattista, L. P. Santos, A. Chalmers: "Accelerating the Irradiance Cache through Parallel Component-Based Rendering", pp. 27–34

back cover:

(from left to right)

Fire simulation: courtesy of Sheldon Tieszen, Sandia National Laboratories.
A. Cedilnik, B. Geveci, K. Moreland, J. Ahrens, J. Favre: "Remote Large Data Visualization in the ParaView Framework", pp. 163–170

Airplane cockpit: Rendered by Abe Stephens, SCI Institute University of Utah, with the Manta Interactive Ray Tracer. 3D data provided by The Boeing Company.
Abe Stephens, Solomon Boulos, James Bigler, Ingo Wald, and Steven Parker: "An Application of Scalable Massive Model Interaction using Shared-Memory Systems", pp. 19–26

Climate modeling simulation: courtesy of Mark Taylor, Sandia National Laboratories.
A. Cedilnik, B. Geveci, K. Moreland, J. Ahrens, J. Favre: "Remote Large Data Visualization in the ParaView Framework", pp. 163–170

International Program Committee

Georges-Pierre Bonneau, Grenoble University, France
Dirk Bartz, University of Tubingen, Germany
Kadi Bouatouch, University of Rennes I, IRISA, France
Montserrat Boo Cepeda, Universidad de Santiago de Compostela, Spain
Alan Chalmers, University of Bristol, UK
Paolo Cignoni, I.S.T.I. - C.N.R, ITALY
Thomas Ertl, University of Stuttgart, Germany
Issei Fujishiro, Tohoku University, Japan
Charles Hansen, University of Utah, USA
Jian Juang, University of Tennessee, Knoxville, USA
Margarita Amor Lopez, Univ. da Coruña, Spain
Kwan-Liu Ma, University of California-Davis, USA
Jamie Painter, BlackMesa Capital, USA
Eric Reinhard, University of Bristol, UK
Bengt-Olaf Schneider, NVIDIA
Han-Wei Shen, The Ohio State University, USA
Claudio T. Silva, University of Utah, USA
Philipp Slusallek, Saarland University, Germany
Rüdiger Westermann, Technische Universität München, Germany
Craig M. Wittenbrink, NVIDIA

Additional Reviewers

Louis Bavoil, University of Utah, USA
Li Chen, University of Tokyo, Japan
Steve Callahan, University of Utah, USA
Erwin Coumans, Sony Computer Entertainment America, USA
Kurt Debattista, University of Bristol, UK
Yoshinori Dobashi, Hokkaido University, Japan
Akio Doi, Iwate Prefectural University, Japan
Nat Duca, Sony Computer Entertainment America, USA
Nathan Fout, University of California-Davis, USA
Vangelis Kokkevis, Sony Computer Entertainment America, USA
Nelson Max, University of California-Davis, USA
Carlos Scheidegger, University of Utah, USA
Jurgen Schulze, University of California San Diego, USA
Simon Stegmaier, University of Stuttgart, Germany
Magnus Strengert, University of Stuttgart, Germany
Bernhard Thomaszewski, University of Tubingen, Germany
Huy T. Vo, University of Utah, USA

Table of Contents

Eurographics Symposium on Parallel Graphics and Visualization - Short Papers (2006)
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

Pipelined Sort-last Rendering: Scalability, Performance and Beyond

Xavier Cavin[†] and Christophe Mion (Inria - Alice)

Abstract

We present in this paper a theoretical and practical performance analysis of pipelined sort-last rendering for both polygonal and volume rendering. Theoretical peak performance and scalability are studied, exhibiting maximum attainable framerates of 19 *fps (volume rendering with back-to-front alpha blending) and* 11 *fps (polygonal rendering with Z-buffer compositing) for a* 1280×1024 *display on a Gigabit Ethernet cluster. We show that our implementation of pipelined sort-last rendering on a* 17*-node PC cluster can nearly sustain these theoretical figures. We finally propose possible enhancements that would allow to go beyond the maximum theoretical limits. This paper clearly shows the potential of pipelined sort-last rendering for real-time visualization of very large models on standard PC clusters.*

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems-Distributed/network graphics; I.3.3 [Computer Graphics]: Picture/Image GenerationViewing algorithms; C.2.4 [Computer-Communication Networks]: Distributed SystemsDistributed applications

1. Introduction and related works

As polygonal and volume datasets become bigger and bigger (see Figure 1), solutions based on sort-last rendering emerge as a mean to visualize them on PC clusters in place of dedicated high-end systems. These solutions include CEI/HP Parallel Compositing API [CEI, HPP] or Sandia Ice-T [MT03, PAR].

In sort-last rendering, the dataset to be displayed is decomposed and distributed across the different nodes of a PC cluster. For each new frame, each node renders a complete image of the data it has been assigned to, using its local GPU. Then it reads back the content of the frame buffer from the GPU to main memory. A parallel image compositing step is then performed to blend all the full resolution partial images into a final image; this step intensively uses the interconnection network to exchange parts of the composed image. Finally, the master node (responsible for displaying the final image to the user) gathers the final image from the slave nodes (which involves transmitting a full resolution image to a single node), and draws it from main memory to the frame buffer of the GPU.

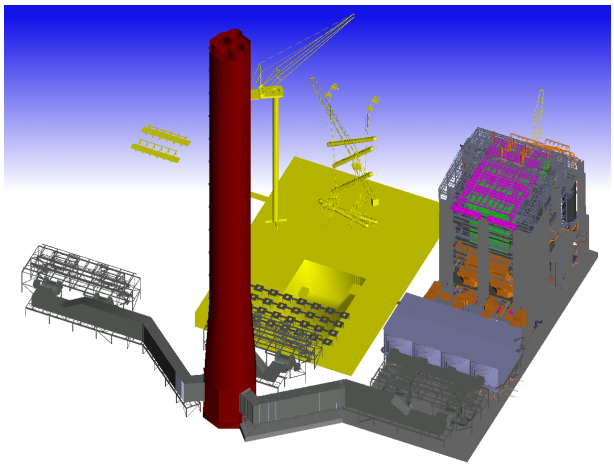


Figure 1: *Real-time visualization of the Power Plant model (data courtesy of UNC-Chapel Hill): the model consists of* $12,748,510$ *triangles and is rendered at* 11 *fps on a* 1280×1024 *display on our* 17*-node COTS PC cluster. No pre-processing has been applied to the initial model, and no acceleration technique has been used to speed-up the rendering (just straightly drawing the triangles).*

[†] Xavier.Cavin@inria.fr

To our knowledge, the latest results for CEI/HP Parallel Compositing API report in [HP 05] 28.2 fps for a $512 \times 512 \times 512$ volume dataset and 20.6 fps for a 28 million triangle model on a 8-node PC cluster with InfiniBand 4x interconnect and a 1280×1024 screen resolution. Similarly, the latest results for Sandia Ice-T report in [HW05] 15 fps on a 264-node PC cluster with InfiniBand 4x (without unfortunately mentioning the screen resolution).

These figures on very high end PC clusters obviously serve as a reference of the obtainable performance of sort-last rendering. However, if we consider Commodity Off-The-Shelf (COTS) PC clusters, the available interconnection bandwidth is much lower compared to Infiniband. For instance, 650 MB/s are reported in [HP 05] for InfiniBand 4x, and have to be compared with the 128 MB/s bandwidth of standard Gigabit Ethernet.

Recently, Cavin *et al.* have proposed in [CMF05] a pipelined implementation of sort-last rendering on a COTS PC cluster. They reported peak performance of 19 fps for volume rendering and 11 fps for polygon rendering on a 5-node PC cluster with Gigabit Ethernet interconnect and a 1280×1024 screen resolution. If we put these figures into perspective with the latest results of the CEI/HP Parallel Compositing API using Infiniband 4x, they obtain 67% and 53% of the performance respectively for volume and polygon rendering, with only 19% of the interconnection bandwidth (not talking about the price ratio). Unfortunately, their experiments were limited to a small scale PC cluster (4 slave nodes), and the scalability of their approach still remains to be proved.

The goal of this paper is first to demonstrate the scalability of pipelined sort-last rendering, and second to suggest possible ways of acceleration. Section 2 of this paper presents a scalability and performance analysis of pipelined sort-last rendering, and applies the theory to practical cases. In Section 3, we present practical experimentations on our 17-node COTS PC cluster with a Gigabit Ethernet interconnect, and we show that the theoretical values can be attained. We conclude in Section 4 and present possible enhancements that could push pipelined sort-last rendering to new heights.

2. Pipelined sort-last rendering analysis

2.1. Theoretical performance evaluation

In this Section, we present a performance evaluation of the pipelined sort-last algorithm proposed by Cavin *et al.* in [CMF05].

According to their notations, the time needed to display a single frame using standard (non pipelined) sort-last algorithm is decomposed as:

$$time = render + read + compose + collect + draw \quad (1)$$

With their pipelined implementation, that overlaps:

- rendering (*render*) and parallel image compositing (*compose*) of the current frame with final image gathering (*collect*) and drawing (*draw*) of the previous frame;
- reading the frame buffer from GPU to main memory (*read*) with parallel image compositing (*compose*);

the time needed to display the same frame is now:

$$time = \begin{cases} compose & \text{if} \quad render \leq compose \\ render & \text{if} \quad render \geq compose \end{cases} \quad (2)$$

If we assume, as in [CMF05], that:

- we use an ideal COTS cluster with no latencies;
- we use one master node and n slave nodes;
- fps is the number of frames per second on a single node;
- xy is the number of pixels of the display, bpp is the number of bits per pixels of the color frame buffer and $zdth$ is the size in bits of the depth buffer;
- b_{op} is the bandwidth in bits per second of the operation op;

and if we assume that the depth buffer is not collected at the end of each frame, then the terms of equation 2 can be rewritten as:

$$render = \frac{1}{fps} \quad (3)$$

$$compose = xy \times (1 - \frac{1}{n}) \times \left(\frac{bpp}{b_{sendandrecv}} + \frac{zdth}{b_{sendandrecvZ}} \right)$$

where $zdth$ can possibly be zero in the case of back-to-front compositing without the Z-buffer.

Note that equation 2 gives the minimum time to render a frame, in the case where overlap is completely attained. If some parts are not overlapped, then this time would be higher. This equation gives us an upper bound of the maximum obtainable frame rate.

2.2. Applying the theory

To get more concrete, we assume in the remaining of this Section that our application uses a RGBA 32 bits color buffer and a 32 bits depth buffer if required. We also assume an ideal PC cluster with a given interconnect exhibiting a point-to-point full-duplex bandwidth of b Gb/s and capable of an infinite aggregate bandwidth. We will come back later in Section 3 on this assumption.

If we consider that the rendering speed is infinite (*i.e.* $render = 0$), then *compose* gives us an upper bound of the maximal obtainable frame rate on our cluster. Figure 2 shows this upper bound for increasing resolutions, both on a Gigabit Ethernet ($b = 10^9$) and on a InfiniBand 4x ($b = 5.2 \times 10^9$) PC cluster, for polygonal (with Z-buffer compositing) and volume (with back-to-front compositing without Z-buffer) rendering.

The latest results for the CEI/HP Parallel Compositing

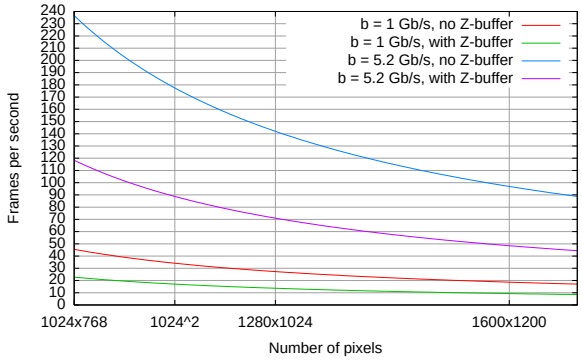


Figure 2: *Theoretical optimal rendering speed of pipelined sort-last rendering in frames per second on a* 17*-node PC cluster, with different interconnects (Gigabit Ethernet* $b = 1$ *Gb/s, InfiBand 4x* $b = 5.2$ *Gb/s), with zero rendering time (*$render = 0$*), with and without Z-compositing.*

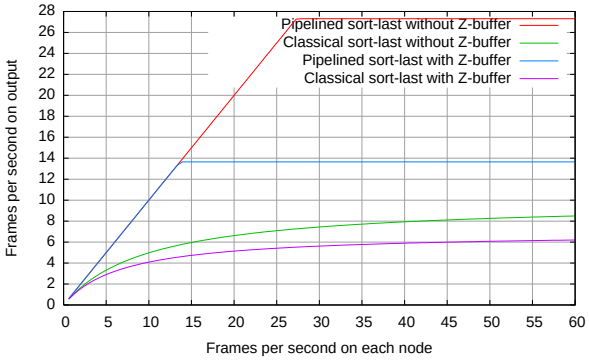


Figure 3: *Evolution of the theoretical optimal rendering speed in frames per second for different nodes rendering speeds on a* 17*-node PC cluster with Gigabit Ethernet interconnect (*$b = 1$ *Gb/s), a* 1280×1024 *display, with and without Z-compositing.*

API reported in [HP 05], *i.e.* 28.2 fps for volume rendering and 20.6 fps for polygonal rendering on a 8-node PC cluster with InfiniBand 4x interconnect and a 1280×1024 screen resolution, are below the theoretical optimal performance of pipelined sort-last rendering, *i.e.* 140 fps and 70 fps. This clearly shows the potential enhancements that pipeline sort-last rendering could bring to their implementation. On the other hand, the 19 and 11 fps reported by Cavin *et al.* [CMF05] on a 5-node PC cluster with Gigabit Ethernet interconnect are closer to the theoretical maximum performance, *i.e.* 27 fps and 14 fps.

It is also interesting to study how the maximal obtainable frame rate varies with different rendering speeds (*i.e.* different values of *render*), as illustrated on Figure 3: the curves compare the theoretical performance of the pipelined sort-last algorithm compared to the classical sort-last algorithm with or without taking into account the Z-buffer. They clearly show the gain of pipelined sort-last rendering as compared to classical sort-last rendering (up to 300% for the case without the Z-buffer).

3. Experimentation

3.1. Experimental setup

We have implemented the pipelined sort-last algorithm on Linux in C using OpenGL, Pthreads and TCP/IP sockets. We have run our experiments on a 16-node COTS PC cluster. Each node is equipped with:

- a bi dual core AMD Opteron (275) 2.4 GHz;
- a NVIDIA GeForce 6800 Ultra 512 MB;

and nodes are interconnected through a Gigabit Ethernet interconnect.

We have performed experimentations both with polygon rendering (with Z-buffer compositing) and volume rendering (with back-to-front compositing without Z-buffer). For polygonal rendering, we have written a very simple OpenGL application that displays polygonal models stored in PLY format. For volume rendering, we have written another OpenGL application that displays volumetric raw data.

3.2. Black screen rendering

Our first benchmarks rely on a "black screen rendering" scheme: nothing is rendered ($render = 0$) and the black images are composited together (without optimization of any kind) and displayed on the master node. This allows to test the maximum attainable framerate on our PC cluster. For any number of slave nodes (from 2 to 16), we obtain the same output rendering speed: Table 1 reports the obtained performance for different resolutions and compares it to the theoretical maximal value. The results are very similar to those reported in [CMF05] on their 5-node PC cluster. They also point out that the overlapping of the different parts of the algorithm is not completely perfect, *i.e.* some parts are serialized.

The scalability of our implementation for PC clusters bigger than our 17-node one only relies on the capability of the interconnect to handle the simultaneous $\frac{n}{2}$ point-to-point full-duplex communications occurring during the parallel image compositing stage between the n slaves of the cluster. Indeed, in our analysis, we have made the assumption that the interconnect supports an infinite aggregate bandwidth. This is clearly and unfortunately not the case. We point out that a lot of care must be taken in the choice of the interconnection switch when building a large graphics PC cluster.

Resolution	Volume	Polygon
1024×768 (theory)	46 fps	22 fps
1024×768 (observed)	**31 fps**	**17 fps**
1280×1024 (theory)	27 fps	14 fps
1280×1024 (observed)	**19 fps**	**11 fps**

Table 1: *Observed and theoretical output performance of our pipelined sort-last implementation for "black screen rendering" (render = 0) on our 17-node PC cluster in the volume (without Z-buffer) and the polygonal (with Z-buffer) rendering case.*

3.3. Case study: the Power Plant model

As a final experiment, we have used our polygonal rendering application to visualize the famous Power Plant model (see Figure 1). This model is composed of $12,748,510$ triangles distributed among several sections. In our application, we do not use any acceleration technique of any kind (no Level of Details, no culling, no impostors, no display list, no vertex array, ...): we simply draw each triangle one after the other by sending its three vertices to the GPU. Moreover, we have performed absolutely no pre-processing on the model. This allows the interactive manipulation of the triangles of the model at constant frame rate, as compared to more sophisticated rendering approaches relying on a static model.

The decomposition of the model across the different nodes is very simple: every single triangle that is read from a PLY file is distributed in a round-robin manner to the given nodes. This ensures an excellent load balancing, since for any point of view, the visible triangles are equally distributed among the nodes. On our 17-node PC cluster, each sub-model (composed of less that a million of triangles) can be rendered at 45 fps on each node. Not surprisingly, the whole Power Plant model can be displayed at the framerates reported in Table 1.

4. Conclusion and future works

In this paper, we have presented an analysis of the performance and of the scalability of the pipeline sort-last rendering algorithm introduced in [CMF05]. Our theoretical analysis allows the authors of related works to compare their observed performance with the theoretical maximum value, with respect to their cluster characteristics. It also shows that pipeline sort-last rendering is a very scalable approach, if one take care in the choice of the interconnection switch, which is a potential bottleneck when the number of nodes in the cluster increases. Finally, we have presented experimentations of our implementation of pipeline sort-last rendering on a 17-node COTS PC cluster. These experimentations have proved the scalability of this approach for a cluster of this size; our theoretical analysis makes us confident about the scalability for larger clusters.

However, the results we have obtained are below the theoretical maximum values, as shown by Table 1. This is due to the fact that some parts of the algorithm are not completely overlapped. In particular, we suspect that the exchanges occurring during the parallel image compositing and the send operations occurring during the final collect operation conflict in some way. We are currently investigating these parts to optimize the overlap. This should help us getting closer to the theoretical limit.

Then, the only way to increase the performance will be to increase the available bandwidth for the communications. For instance, Figure 2 shows the theoretical frame rate that could be achieved using an InfiBand 4x interconnect. The drawback of InfiniBand is obviously its price, as compared to classical Gigabit Ethernet. We are currently investigating another (cheaper) way of increasing the available bandwidth: each node of our cluster is actually equipped with four Gigabit Ethernet attachments, which gives us a potential bandwidth of 512 MB/s, to be compared with the 650 MB/s of InfiniBand 4x. Our future works include finding solutions to make a full usage of this potential bandwidth.

A success in both directions (maximizing the overlapping and increasing the bandwidth) will give a scalable sort-last rendering solution capable of very high framerates with high resolutions.

References

[CEI] CEI Ensight Parallel Compositor. `http://www.ensight.com/`.

[CMF05] CAVIN X., MION C., FILBOIS A.: COTS cluster-based sort-last rendering: Performance evaluation and pipelined implementation. In *Proceedings of IEEE Visualization 2005* (2005).

[HP 05] HP VISUALIZATION TEAM: Compositing using COTS components, 2005. `http://www.hp.com/techservers/hpccn/downloads/HP-Compositing-public.pdf`.

[HPP] Advanced visualization collaboration. `http://www.hp.com/techservers/hpccn/sci_vis/`.

[HW05] HIGHAM D., WYLIE B.: Sandia National Labs achieves breakthrough performance using NVIDIA technology for scientific visualization, 2005. NVIDIA press release, `http://www.nvidia.com/object/IO_19962.html`.

[MT03] MORELAND K., THOMPSON D.: From cluster to wall with VTK. In *Proc. of IEEE 2003 Symp. on Parallel and Large-Data Visualization and Graphics* (2003).

[PAR] Paraview - parallel visualization application. `http://www.paraview.org/`.

Eurographics Symposium on Parallel Graphics and Visualization - Short Papers (2006)
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

A Simple, Distributed Rendering Toolkit for Multi-Screen, Rendering Clusters

A. Tetzlaff[1] C.-A. Bohn[1] T. F. Horn[2]

[1]Wedel University of Applied Sciences
[2]Planetarium Hamburg

Abstract

We present a cluster-based rendering system for driving arbitrary multi-screen environments. Essentials of the system are that there is no sophisticated synchronization hardware required; applications can be developed on desktops and then are able to be copied on nearly any of such environments; the number of rendering nodes is arbitrary scalable for driving any required resolution or any number of projection screens.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Distributed Rendering Systems

1. Introduction

In recent years, with the development of graphics hardware users' demand in highly advanced graphics output became a crucial issue. Since display sizes and resolutions seem to meet their physical limits, the natural way to overcome this limitation is to simply increase the number of displays and combine them physically in a way that they imitate one large screen.

Some early attempts are the well-known double-screen desktop environments. More recent technologies include *Virtual Reality* Technologies (i.e., *head-mounted displays* [FMHR87]), multi-screen training- (i.e. flight-, drive-) simulators, multi-projector screens like *CAVE*-like environments [CNSD93], high-resolution *Power Walls* [HH99], and dome-like cinemas like *IMAX* theaters and planetariums.

Vital challenge in developing applications for multi-screen technologies is to handle the underlying rendering hardware. Whereas two-screen environments often can be realized by a single computer feeding one single graphics card, more sophisticated installations often base on the use of several PCs. Developers have to make strong efforts for parallelizing the application and the rendering.

In this work we describe a system applicable to common multi-screen environments, which is tested in a concrete installation — the Planetarium Hamburg [pla]. This planetarium consists of seven rendering PCs driving seven projectors forming together one spherical projection dome.

Figure 1: *The Planetarium-Hamburg — test case for the presented approach.*

Usually, access to soft- and hardware of planetariums for experimental applications is very restricted. Often they only run commercial productions which are pre-rendered and just "played". Here, parallelization is realized by a kind of a "parallelized video player". Thus, incorporating own (*OpenGL*-) applications must happen by avoiding massive infringement on the daily operational demands of such an environment and on secrecy issues from its hard- and software supplier.

In this work, we show how such a rendering system can be realized without any change in the aimed target hardware, without touching the software configuration, and without a vast amount of work put into parallelization issues of render-

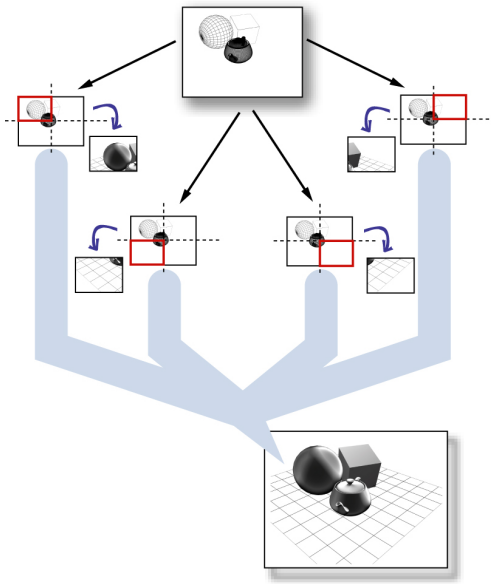


Figure 2: *Partitioning of one screen into several sub-screen connected to a single rendering node each. Nodes are separate programs, according to the master-slave clustering model, running on the same or different platforms.*

ing tasks. Any pre-compiled *OpenGL*-application (the binaries) just need to be copied and executed on each rendering node.

Several alternative software packages for distributed rendering are available, only mentioning some of them like *Chromium* [HHN*02], *OpenSG* [osg], the *CAVE library* [cli], *VR Juggler* and *Net Juggler* [jug], or *Avango* [avg]. A great comparison of these systems can be found in [vrc].

In the following we shortly outline the basics of our system, present its vital advantages compared to classical approaches and finally we will conclude with a summary and show two example applications.

2. Approach

Porting applications written for *one* graphical output to a multi-screen environment is not a trivial task. Considering the simplest case — one graphics card on one computer drives two screens — the programmer must handle two framebuffers. Going steps further would become harder — two graphics cards need a parallelization of the rendering tasks, two computers with own graphics hardware requires additional parallelization of separate processes, additional synchronization and network communication.

2.1. Structure of the Rendering Cluster

The underlying system consists of a set of rendering nodes which create the final picture by rendering each a certain cut-out of the whole spherical screen (see Figure 2). The nodes configuration can be classified according to [SWNH03] as a typical *master-slave* cluster. The complete scene data is available on every rendering node. The animation and rendering status of each of them is controlled by one master node which is also responsible for tracking and distributing user interaction. In this work the term “node” is not connected to a certain hardware platform, moreover rendering nodes and the master node are just separate programs which may run on different platforms but also as concurrent tasks on the same platform. “Distributing an application” simply means taking the *OpenGL* program and executing it *n*-times as *n* separate tasks on one or more separate rendering platforms.

After the start each rendering node first connects to the master node. The master node then triggers the rendering and “supervises” the rendering nodes in the following time. Input events arising at the master’s periphery for, i.e., modifying the viewing parameters, are passed to each rendering node. Only in case of user interaction like key strokes or mouse movements the according event data is transfered. The amount of this data is negligible, thus we realized the transfer by using common network transfer technology — a 100 MBit network, which even was the only available at our test environment at the planetarium.

2.2. Synchronization

The separate pipelines must display their image parts in a synchronized manner. In our application we are satisfied with a maximal delay of one frame on separate rendering nodes. Thus, the approach is capable of driving all non-stereo multi-screen systems, like in our application the Planetarium-Hamburg, all active stereo systems with a specialized hardware synchronization like [dig], or all passive stereo systems.

The basics of our synchronization are as follows. In advance to the start of the actual application all nodes of the cluster have to connect to the master via *TCP*. The master listens for the rendering nodes at a predefined port to receive their local configurations. Then it provides them with his demand in rendering and simulation speed in terms of a certain value of *frames per second (FPS)*. Up to now the transfer is not time-critical and thus delivered through *TCP*.

To start the animation the master sends a start-command via *UDP* to all nodes. Now, by choosing *UDP* we abandon the flow control overhead of *TCP* enabling short transport times for time-critical data like the start-command. These transport times virtually are negligible. Then rendering nodes start at a predefined status — synchronized except for the delay the UDP transfer causes. Through several tests

we proved that the delay is clearly below one frame and thus negligible.

After initial synchronized triggering the nodes compute the animation parameters and finally render their frustum of the scene. Herewith, every node tries to maintain the frame rate demanded by the master node, i.e. if the master wants a 40 frames per second rate, the rendering node waits until a full 1/40 of a second went by until the next frame is generated.

In other words, our synchronization is realized by the internal clocks of each rendering node. This is uncommon but satisfies our application conditions excellently. It leads to the fact that we virtually do not have synchronization transfer overhead.

Of course a lack of synchronization may arise due to the following reasons.

- The master's distribution of the current frame number at the start is delayed.
- The clock rate on separate nodes differ.
- Some nodes cannot achieve the requested rendering and animation speed — they "arrive too late" after one rendering task.

To avoid the first two cases "re-synchronization"-packets are sent every few seconds. In all cases the according nodes react in the same manner. They try to catch up to the correct frame number by skipping the rendering part and just proceeding with the integration part of the animation.

This minimal amount of transfer makes it possible to scale the number of rendering nodes arbitrary without reducing the rendering performance.

2.3. Handling User Events from the Master Node

Events recorded by the master node have to be delivered to the rendering nodes. Herewith, it must be guaranteed that all nodes react on events at the same time — the same frame number — otherwise the animation diverts on different nodes. To avoid that, the master node collects and serializes all events of the current frame, marks this event set by the current frame number an distributes it via *UDP* packets to the rendering nodes. Each of them stores the dataset temporarily and triggers the contained events not until it reaches the frame number which the master attached at the event set. Thus a consistent animation is guaranteed even if nodes lag a frame due to their local rendering resources. By incrementing the assigned frame number before sending an event set it is guaranteed that even nodes which are one frame ahead will register the events at the right frame. Although this results in a minimal latency of user interactions it is preferable compared to an inconsistent animation.

In our desktop environment and also in the planetarium's PC cluster, it has been shown that variations of up to two frames do not affect the visual quality.

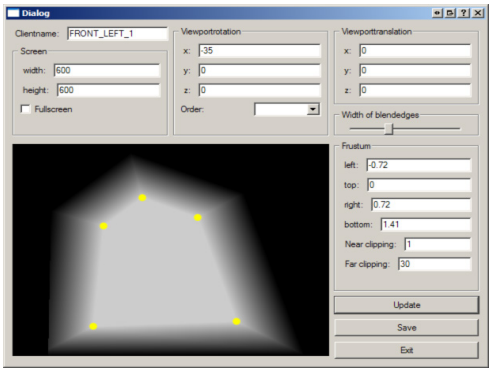


Figure 3: *The user interface to define viewing and blending parameters of overlapping projection areas illuminated from different rendering nodes.*

2.4. Edge Blending

Projections physically combined to large displays typically overlap at the borders to avoid cracks in the whole projected area. At these overlapping areas the generated image would appear brighter if it is not dimmed accordingly at each participating projector output. Since only few projectors have a *hardware edge blending* capability to realize this adaption, we added edge blending by software implemented through using the blending facilities available on common graphics cards.

2.5. Rendering Node Configuration

The basic configuration — parameters like shape, orientation, and position of the frustum — of each node of a multi-screen environment is specified by one *XML*-file per node. It is parsed by the rendering application at startup and sent to the master application when logging in. Here, also the edge blending cut-out is defined by an arbitrary polygon which can be defined through a graphical configuration interface (see Figure 3).

3. Results and Summary

Our proposal enables the user to write multi-screen environment applications fairly easy and straight-forward.

- *Implementation:* The user simply writes an *OpenGL* program on a usual computer with one graphics pipeline. Then, porting the application to the multi-screen environment means to copy it to all participating rendering nodes only — not even another linkage is necessary.
- *Synchronization:* Synchronizing the rendering nodes is accomplished every few seconds. Inbetween, synchronized rendering is guaranteed through the internal clocks of each node. With this attempt, only negligible data exchange between the master and the rendering nodes arises,

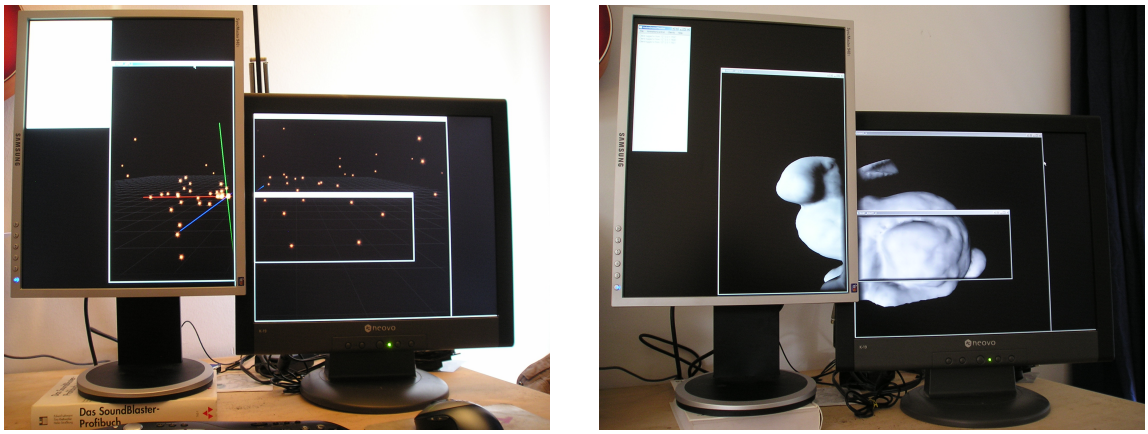


Figure 4: *Test applications (a particle system and a bunny model) of the presented system. Several separate rendering processes running on each of two PCs.*

which makes it possible to realize a rendering cluster of a theoretically unlimited size. Another advantage is that the system does not need sophisticated network hardware — not even a gigabit network — to guarantee synchronization between master and rendering nodes.

- *Test case:* We tested our software developed on a single screen desktop PC in a professional, commercial environment. It worked from scratch without any re-compilation proving its portability, simplicity, and robustness.

Figure 4 exposes two example renderings of a simple particle system and the well-known bunny model. The software of each of the shown implementations consists of just about 300 lines of source code. By using *Qt4* [tro] for high-level access to GUI, network, and threads, the framework is capable of being compiled under Windows and Linux. Adaption to any other hardware environment (like *Cones*, *CAVE*s, or *Domes*) only requires the modification of the frustum and the edge blending parameters on each rendering node. The development process of an application may be completely decoupled from the target-environment.

References

[avg] http://www.avango.org.

[cli] http://www.vrco.com.

[CNSD93] CRUZ-NEIRA C., SANDIN D. J., DEFANTI T. A.: Surround-screen projection-based virtual reality: the design and implementation of the cave. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1993), ACM Press, pp. 135–142.

[dig] http://www.digital-image.de.

[FMHR87] FISHER S. S., MCGREEVY M., HUMPHRIES J., ROBINETT W.: Virtual environment display system. In *SI3D '86: Proceedings of the 1986 workshop on Interactive 3D graphics* (New York, NY, USA, 1987), ACM Press, pp. 77–87.

[HH99] HUMPHREYS G., HANRAHAN P.: A distributed graphics system for large tiled displays. In *VIS '99: Proceedings of the conference on Visualization '99* (Los Alamitos, CA, USA, 1999), IEEE Computer Society Press, pp. 215–223.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM Press, pp. 693–702.

[jug] http://www.vrjuggler.org.

[osg] http://www.opensg.net.

[pla] http://www.planetarium-hamburg.de.

[SWNH03] STAADT O. G., WALKER J., NUBER C., HAMANN B.: A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003* (New York, NY, USA, 2003), ACM Press, pp. 261–270.

[tro] http://www.trolltech.de.

[vrc] http://chromium.sourceforge.net/doc/llnlcopy.html.

Eurographics Symposium on Parallel Graphics and Visualization - Short Papers (2006)
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

Hardware for a Ray Tracing Technique Using Plane-Sphere Intersections

Y. Kaeriyama[1], D. Zaitsu[1], K. Komatsu[1], K. Suzuki[1], N. Ohba[2], and T. Nakamura[1]

[1]Graduate School of Information Sciences, Tohoku University
[2]IBM Research, Tokyo Research Laboratory, IBM Japan, Ltd.

Abstract

Although image synthesis based on the global illumination model can generate high-quality photo-realistic images, it requires a large number of calculations. To accelerate the ray-object intersection test, this paper presents a high speed point sampling method by utilizing a plane and a bounding sphere. The method effectively processes a series of rays radiated from the same point. We propose a hardware system, which parallelizes the calculations of a series of rays.

We evaluated the performance of the proposed system by software simulation. The simulation results show that the proposed system running at 100MHz performs the intersection test about 100 times faster than the commodity PC with 3.4GHz Pentium 4.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture

1. Introduction

Recently, computer graphics has been widely used for the design of engineering products and architectural structures. In these fields, a high-speed high-quality image generation method is crucial for the success. Ray tracing is one of the image synthesis techniques based on the global illumination model. Although it gives photo-realistic images, it requires huge computational time.

There have been many hardware systems for ray tracing proposed so far. In [SWS02], a BSP tree is prepared for objects and the coherency of rays is utilized. Later it has been implemented in an FPGA prototype [SWW*04]. 3DCGiRAM, which integrates functional memories with processing elements for generating images, has been proposed in [KSS*01]. It uses virtual 3D intelligent memories allocated for divided object spaces, and performs the ray-object intersection tests in a fully distributed manner. However, it is effective only for walk-through movies. In [FR03], image synthesis hardware based on a hierarchical bounding box structure has been proposed.

The objective of our research is to build a high-speed ray tracing system. This paper focuses on the intersection tests between the objects and the rays radiated from the same point. We present an intersection test method, which reduces the number of ray-primitive intersection calculations by using bounding volumes and 'ray planes'. We also show the prototype system implemented on an FPGA board.

2. High-Speed Intersection Test

2.1. Intersection Test Using Plane and Bounding Sphere

Point sampling is one of the methods for solving the rendering equation. It randomly casts stochastic rays, and therefore it requires a huge number of calculations. In particular, the intersection test between cast rays and objects requires many calculations, and is often a critical path of the image generation process.

We assume that the first rays are cast to their respective pixels, and it intersects at a particular surface. When a certain ray requires radial sampling at the direction of normal, the ray is divided into multiple rays. In the situation that the rays extend radially from the same point, it is necessary for our proposed techniques to ensure that these rays pass through a grid point on a plane.

Our intersection method using orthogonal planes(ray planes) instad of individual rays is:

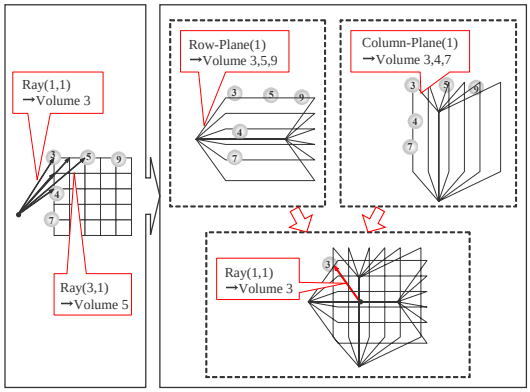


Figure 1: *Intersection test using planes and bounding volumes. This figure illustrates the basic idea of the proposed method. Ray(1,1) is represented by two planes, Row-plane(1) and Column-plane(1). The intersection test for Ray(1,1) is done by two steps, that is, the intersection tests for Row-plane(1) and Column-plane(1). If both of Row-plane(1) and Column-plane(1) intersect Volume 3, Ray(1,1) intersects Volume 3.*

1. Generate the plane containing the rays for each row/column.
2. Create bounding volumes containing primitives.
3. Calculate the intersection between the planes and bounding volumes(Figure 1). A ray-volume intersection is given when both the two planes containing a certain ray intersects bounding volume. Then, every primitive in the bounding volume is checked if it intersects with the concerning ray or not.

We call this method RP(Ray Plane) method. It greatly reduces the number of calculations between rays and volumes. Assume that the rays are cast to $M \times N$ grid points. The conventional method has to perform the intersection test for $M \times N$ rays; the RP method, on the other hand, does the test for $M + N$ planes.

In addition, to make the intersection test faster, we use a sphere as the bounding volume. The intersection test between a plane and a sphere can be done by the calculation of the distance between the plane and the center of the sphere.

Note that the idea is applicable to various ray tracing operations, such as sampling of the first rays, sampling at Lambertian surfaces, and calculation of shadow rays for square light sources. To reduce the loss of accuracy, a weight factor is multiplied to respective rays.

2.2. Data Structure for Bounding Spheres

There are several structures to hold a set of bounding spheres. In this paper, we consider two structures. One holds the spheres uniformly. We call it 'flat structure'. The other method is to make a tree to hold spheres as shown in Figure 2. While the tree can reduce redundant intersection tests by traversing it, it tends to cause complicated memory access patterns because the tree is generally implemented by pointer linked fractions of data. On the other hand, while the flat structure generates linear memory access patterns, it cannot reduce the number of intersection tests by itself. Taking account of the feature of throughput-oriented DRAM devices of today and the reduction of the intersection tests by the method in section 2.1, the flat structure seems to be well-balanced solution. We discuss the effectiveness of the data structures in the following section.

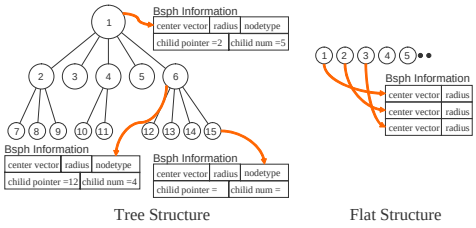


Figure 2: *Data structure*

(a) Cubes (b) Roses (c) Z33

Figure 3: *Test Scenes*

3. Effectiveness of Data Structures

In order to evaluate the intersection methods and data structures, we measured the execution time of image synthesis on software implementation. We made ray tracing programs, one based on a conventional ray-bounding-sphere method and the other based on the RP method. Both the flat structure of spheres and the tree structure of them were prepared for both programs, where the maximum depth of the tree was 3. Three test scenes shown in Figure 3 were generated and their conditions are shown in Table 1. Note that soft shadows by square light sources and reflection on Lambertian surfaces are calculated, as you can see in Figure 3. We used an IBM/AT commodity PC with 3.4GHz Intel Pentium4 and 2GB memory. The programs were written in Visual C++ and executed on Windows XP Professional.

The results are shown in Table 2. In the case of the flat data structure, the RP method is much faster than the conventional one. However, in the case of the tree data structure, the execution time of RP is larger than the conventional

Table 1: *Scene Conditions*

Scene	Cubes	Roses	Z33
# first rays	256x256	256x256	256x256
# bounding spheres	177	2612	23638
# triangles	708	11396	61828
# square light sources	1	2	1
# light source sampling	16x16	16x16	16x16
second ray depths	1	0	0
# lambert sampling	16x16	16x16	16x16

Table 2: *Software execution time (sec.)*

	Bsph	Cubes	Roses	Z33
Conventional	Flat	603	11217	44510
(Ray-Bsph)	Tree	441	1340	1240
RP method	Flat	99.9	850	3510
	Tree	53.2	685	2372

one for the Z33 scene. The reason is described as follows. RP method reduces the number of tests by combining multiple rays into a plane, as shown in case of Cubes and Roses. This is true for Z33, too. However, when the tree structure is used, the number of tests is larger than the conventional method, because a plane consisting of a series of rays intersects with much more spheres than an individual ray. On the other hand, in the conventional method, each ray can traverse the tree individually with avoiding redundant intersections.

4. Dedicated System for Fast Ray Tracing

4.1. Strategy of Hardware Design

Highly parallel processing is indispensable for the high-performance ray-tracing hardware that achieves the significant speed-up in comparison with a commodity general purpose processor. Parallel processing systems, however, often suffer from the bandwidth limitation in the data transfer between the processing elements and the memory device. The RP method with the flat structure of bounding spheres is suited for a highly parallel system, which has multiple processing units. The reasons are:

1. It does not traverse the link pointers. All the processing units can process the same bounding sphere at the same time. Therefore, only one data fetch is required to process each bounding sphere.
2. It reads the object data from the memory in highly sequential access patterns which are easily optimized by the burst transfer mode of SDRAM.

4.2. Estimation of Data Transfer

In this sub-section, we estimate the amount of data transfer for generating an image in the RP method. Here, we use the parallel processing model with 32 intersectors, which performs the intersection tests, connected to a shared DRAM. Taking account of the fact that the data size of ray planes is much smaller than that of bounding spheres, we assume that all the data for planes are distributed to the intersectors prior to the start of the calculation, and the bounding spheres are delivered form the DRAM one by one. Each intersector calculates whether or not the plane intersects with the bounding sphere. We assume that the number of cycles to fetch the bounding sphere data form the SDRAM is eight for the first data of the burst read and one for subsequent data. The test scenes for the estimation are the same as those used in the previous section.

Table 3: *Volume and Number of cycles for data transfer*

		Cubes	Roses	Z33
Volume of	Tree	1.72	21.7	63.8
Data Transfer (TB)	Flat	0.241	3.83	14.4
Number of cycles for	Tree	0.770	2.29	12.5
Data Transfer (10^9)	Flat	0.225	0.344	1.29

Table 3 shows the amount of data transfer and the number of cycles for reading sphere data, respectively. From the results, we can see that the flat structure for the RP method reduces the amount of data transfer dramatically. The reason of this great reduction is that, in case of the flat structure, all the intersectors can use the same data at every intersection test, because all the planes are checked for intersection with all the bounding spheres. Therefore, using the flat structure, every bounding sphere is fetched from the DRAM and can be broadcast to all the intersectors regardless of the number of intersectors.

4.3. Structure of the Proposed Hardware

In this section, we propose a hardware implementation of ray-tracing system based on the RP method called RAPLAS (Ray tracing Architecture based on PLane and Sphere intersection). The operations of bounding spheres and planes are performed in a massive parallel manner to achieve significant speed up.

A block diagram of RAPLAS is shown in Figure 4. It consists of two major blocks, Bounding Sphere Intersection Block and Object Primitive Intersection Block. The Bounding Sphere Intersection Block executes the intersection tests of bounding spheres and planes containing a series of rays. The Object Primitive Intersection Block executes the intersection tests of object primitives and respective rays.

The data of the bounding spheres are fetched form the Bsph MEM, and distributed to the Plane-Bsph intersectors. Each Plane-Bsph intersector performes the intersection test and generates a check flag showing whether it finds the intersection or not. The check flags from all the Plane-Bsph intersectors are ANDed to formulate the check flag of each ray. The Object Primitive Intersection Block performs the intersection tests between the primitives only in the bounding sphere with the check flag set and the corresponding ray.

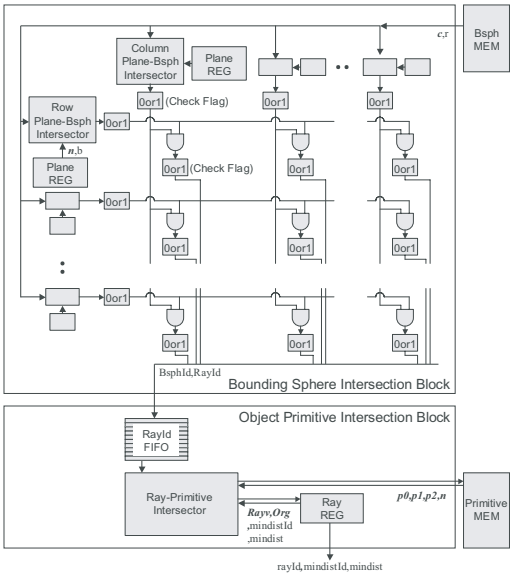


Figure 4: *Block diagram of RAPLAS*

Table 4: *Estimation of hardware execution time(sec)*

	Cubes	Roses	Z33
Software (fastest)	53.2	685	1240
Hardware (estimation)	5.48	6.3	12.9
Performance Increase	x 9.71	x 108	x 96.1

5. Performance Evaluation of the Proposed Hardware

5.1. Software Simulation

We did software simulations to evaluate the proposed hardware system. We assume the following parameters for the simulation. RAPLAS core runs at 100MHz. Data fetching from the memory is fast enough with zero delay. Plane-Bsph intersectors and the Ray-Primitive intersector are pipelined. In this experiment, in order to process 16x16 rays at a time, 32 Plane-Bsph intersector pipelines and 1 Ray-Primitive intersector pipeline are prepared, and the throughput of the both pipelines are 1.

The results are shown in Table 4. According to the results the proposed hardware running only at 100MHz, achieves ten to a hundred times of speed up compared with a general-purpose processor, which has 30 times higher operating frequency.

5.2. FPGA Implementation

Following the simulations, we are implementing RAPLAS using an FPGA board shown in Table 5. Since the Xilinx Virtex4-SX FPGA has hardcore DSPs, it is suited for the prototype implementation of RAPLAS.

We have started the implementation with the Bounding Sphere Intersection Block consisting of the Plane-Bsph Intersectors and the AND circuits, which processes 4×4 rays at a time with a pipeline of 11 stages. It consumes 33 % of logic cells and 96 DSPs in the FPGA. We consistently use 24bit floating point data format consisting of 7 bit exponential and 16 bit significand. We have confirmed that the FPGA is running at 100MHz, and the Bounding Sphere Intersection Block mentioned above can achieve the performance at the delay of 11 cycles.

Table 5: *Implementation target*

Board	Avnet MB Development Board	
FPGA	Device	Xilinx XC4VSX35
	Logic Cells	34560
	DSP	192
	Block RAM	3456Kbit

6. Conclusions and Future Works

In this paper, we have proposed a parallel and dedicated hardware for high speed ray tracing based on planes and bounding spheres. The proposed hardware, RAPLAS achieves approximately 100 times speed up over a commodity PC when it runs at 100MHz. We will study the effects of approximate sampling on image quality, and implement and evaluate the whole RAPLAS system including the Object Primitive Intersection Block.

References

[FR03] FENDER J., ROSE J.: A high-speed ray tracing engine built on a field-programmable system. In *IEEE International Conference on Field-Programmable Technology* (2003), pp. 188–195.

[KSS*01] KOBAYASHI H., SUZUKI K., SANO K., KAERIYAMA Y., SAIDA Y., OBA N., NAKAMURA T.: 3dcgiram: An intelligent memory architecture for photo-realistic image synthesis. In *Proc. International Conference on Computer Design: VLSI in Computers & Processors* (2001).

[SWS02] SCHMITTLER J., WALD I., SLUSALLEK P.: Saarcor - a hardware achitecture for ray tracing. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware* (2002), pp. 27–36.

[SWW*04] SCHMITTLER J., WOOP S., WAGNER D., PAUL W. J., SLUSALLEK P.: Realtime ray tracing of dynamic scenes on an fpga chip. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware* (2004), pp. 95–106.

Eurographics Symposium on Parallel Graphics and Visualization - Short Papers (2006)
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

Accurate Workload Estimation for Fast Parallel RMT

Timothy S. Newman [1] and Wenjun Ma[1]

[1] Department of Computer Science, University of Alabama in Huntsville, Huntsville, AL 35899, USA

Abstract

A model that accurately estimates isosurface extraction effort for the Regularised Marching Tetrahedra (RMT) [TPG99] isosurfacing is introduced and evaluated. The model enables parallel, load-balanced isosurfacing. The performance of the model versus alternatives is evaluated based on computational speed and accuracy. The performance of parallel RMT using the model is also exhibited.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Graphics data structures and data types

1. Introduction

In this paper, a work estimation model that allows fast production of compact isosurfaces is introduced and evaluated.

Isosurface extraction and display is one of the commonly-used methods to visualize volumetric data such as Computerized Tomography (CT) and Magnetic Resonance Imaging (MRI) data. The aim of isosurfacing is to locate a contour of constant value in the data. Isosurface extraction in volumetric data usually entails creating a mesh of triangles that represents the isosurface. Often, the well-known Marching Cubes (MC) algorithm [LC87, Nie03] is the means for determining the isosurface. In MC, the dataset is processed through a sequential forward march over the cubes.

As originally presented, MC had the defect of possibly producing isosurfaces which contained "holes" (i.e., gaps in the isosurface). The hole can cause a "void" to be observed in an isosurface rendering. The Marching Tetrahedra (MT) [PT90] variant on MC is one way to generate a hole-free isosurface. In the standard MT, each cube is first subdivided into five (or six) tetrahedra. Then, the isosurface is formed in each tetrahedron of each cube. Although an MT isosurface has no holes, the standard MT subdivision scheme produces tetrahedra whose shapes differ, which leads to greater variance in facet size than in MC [CP98, TPG99]. Another disadvantage of the MT is that its isosurfaces usually contain twice the triangles produced by applying Marching Cubes [SB00]. The increase in triangles is a modest challenge to rendering hardware. It is a more significant challenge for storage and transmission, for example in distributed isosurfacing. MC also executes about twice as quickly as MT [SB00].

The body-centered cubic (BCC) subdivision [CP98] is an alternate means for subdivision that can overcome the MT's differing tetrahedra shape problem. The BCC forms tetrahedra that span cubes. Each tetrahedron has two vertices that are at the center of a cube. The other vertices are at the lattice points. To apply BCC-based MT, it is thus necessary to interpolate a data value at each cube centroid. In Figure 1, we illustrate the BCC subdivision for one cube. In the top left of the figure, a cube and the tetrahedral edges for the tetrahedra that intersect the left face of the cube are shown. Each cube face is intersected by four tetrahedra in the BCC subdivision. The tetrahedra that intersect the left cube face are highlighted at the right and bottom of the Figure 1. Advantages of MT using the BCC subdivision include a more uniform triangulation of the isosurface and less sharp edges in the triangulation [TPG99].

Isosurfacing using a BCC-based MT approach maintains the MT disadvantage of producing many triangles and running slowly. The Regularised Marching Tetrahedra (RMT) [TPG99] is an approach designed to overcome the problem of too many triangles. RMT couples BCC-based MT with mesh simplification based on the popular vertex clustering. RMT has been reported to produce 70% less triangles than the MT [TPG99] and perhaps 40% less triangles than the MC [TPG99]. Thus, RMT is attractive from the standpoints of data storage and isosurface facet regularity. In

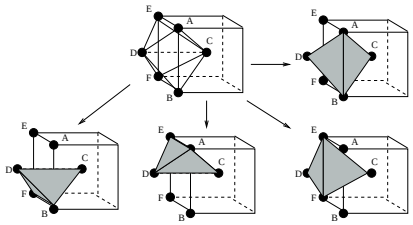


Figure 1: *Body-Centered Cubic (BCC) subdivision of cube and the tets that intersect one face.*

this paper, we introduce a model that provides the means to perform RMT-based isosurfacing efficiently in cluster computation environments. This work is, to our knowledge, the first report of a parallel RMT. Parallel RMT is attractive in cluster computation environments, especially for isosurfacing of large datasets, since the network that interconnects the processors is typically a processing bottleneck for applications that involve even only a modest amount of synchronizing communication.

This paper is organized as follows. Related work is described in Section 2. Section 3 describes the new model and its use in parallel isosurfacing. The scheme's load-balancing mechanism is compared versus other mechanisms in Section 4. Results from applying the scheme are described in Section 5. The conclusion is in Section 6.

2. Related Work

Isosurface extraction can be time-consuming and produce a large mesh, especially when the isosurfacing is via MT and applied to large datasets.

2.1. Parallel Isosurfacing

Even for small datasets, a series of isoqueries are usually necessary in data exploration, making fast isosurfacing an advantage. Many parallel isosurfacing approaches have been described. The approaches for multi-processing environments involve dividing the dataset among the processing elements, with each processing element extracting the isosurface in the portion of the volume assigned to it. Finally, the isosurface pieces are combined together and rendered. Various schemes have been used to balance the workload among the processors, such as static load-balancing methods that assign equal regions of space (e.g., equal numbers of layers [AD03]) to each processor and dynamic load-balancing methods that adjust region assignments during processing (e.g., [MN95]). Although most of the approaches have utilized Marching Cubes, dynamic load-balancing methods for MT have also been described [CFSW01, GR00]. For cluster computation environments, the large number of triangles produced by parallel MT approaches produces a bottleneck in the final stage of processing (i.e., gathering the isosurface facets into a unified representation for rendering).

2.2. Vertex Clustering and RMT

Many strategies for mesh simplification have been described. One such strategy is vertex clustering (e.g., [LT97]). In vertex clustering, if a group of mesh vertices are very close together, they are merged into a single vertex. When applied to an isosurface mesh, vertex clustering can yield a mesh with a reduced number of facets. In the RMT, a vertex clustering component is embedded in the isosurface extraction processing. Vertex clustering on meshes can change mesh topology which is not desirable for isosurface meshes. Thus, in the RMT, nearby vertices are only clustered if doing so would allow preservation of certain surface topology features. Specifically, RMT does not cluster if any of three types of features occur (e.g., RMT will not cluster across a hole). When vertices are clustered, the replacement vertex position is at the weighted mean position of the eliminated vertices. The weights are based on estimates of curvature, which promotes preservation of corner features [TPG99].

RMT processing involves a series of operations on each cube: (1) interpolation of the center point's data value, (2) determination of which data points equal or exceed the isovalue, (3) determination of which (if any) tetrahedra are intersected by the isosurface, (4) computation of the facet vertices, (5) vertex clustering (if appropriate) and (6) formation of the facets.

3. Static Load-Balancing

In our parallel RMT approach, we utilize static load-balancing with workload divided among the CPUs based on an accurate load estimation model. An advantage of static load-balancing over dynamic load-balancing is that there is no overhead such as processor synchronization during execution to balance out load. However, for a static approach to well utilize the processors, workload must be estimated accurately. For overall performance to be acceptable, the estimation must also be fast.

The model that we use for workload estimation is linear in the number of tetrahedra, N, the number of sample points whose values exceed the isovalue, A_{vh}, and the number of tetrahedra edges intersected by the isosurface, A_e. Other factors (the number of sample points connected to an intersected edge, A_{vs}, and the number of active cells, A_t) were also considered, as described in Section 4, but were rejected due to ill-suitedness. A linear model was considered since there are no inter-cell or inter-edge comparisons in basic isosurfacing. The model used is very accurate and can be computed quickly. We employ the model in static load-balanced RMT to estimate work in each *blocklet* of the dataset. A blocklet is a small set (we use $8 \times 8 \times 8$ regions) of contiguous cubes. The model used takes isosurfacing computation time as the measure of workload. For the three-parameter model, (N, A_{vh}, A_e), used, the time measure, T, is

$$T(N, A_{vh}, A_e) = aN + c_1 A_{vh} + c_2 A_e.$$

In practice, the weights a, c_1, and c_2 are determined by applying least squares on the execution times and parameter measures of some set of reference isoqueries.

After workload estimation, the blocklets are divided among the CPUs based on workload; each CPU is assigned an approximately equal amount of work. The isosurface facets constructed by each CPU are gathered on one "master" CPU and then rendered to the display. Due to the use of the BCC subdivision, each blocklet contains many partial tetrahedra that lie along the blocklet boundary. To avoid processing such tetrahedra in both blocklets that contain them, we have the i^{th} blocklet's processing handle the tetrahedra that straddle the blocklet's right, back, and top sides (but not the tetrahedra that straddle the blocklet's other sides). An extra layer of center points is also added as padding to each of the left, front, and bottom bounds of the dataset. Moreover, since the vertex clustering component of the RMT must consider 14 neighboring vertices for every sample point, the neighbors of those points in each blocklet must be packaged with the blocklet. In the parallel realization, the duplication of data near blocklet boundaries involves some unavoidable duplication of some work for adjacent blocklets assigned to different processors.

(N, A_{vs})	(N, A_{vh})	(N, A_e)	(N, A_t)
243	715	134	149

Table 1: *Avg. relative error (%) in workload estimation models, 2-parameter models (Lobster Dataset)*

$(N, \ldots$	$A_{vs}, A_e)$	$A_{vh}, A_e)$	$A_{vs}, A_t)$	$A_{vh}, A_t)$	$A_e, A_t)$
	17.0	14.1	17.5	14.8	18.4

Table 2: *Avg. relative error (%) in workload estimation models, 3-parameter models (Lobster Dataset). Notation here meant to indicate N is a parameter of each model.*

Figure 2: *Isosurface extracted from Lobster Dataset using RMT at* $\alpha = 100$

4. Model Evaluation

Previous work has suggested that MC isosurfacing effort is a function of the number of cells and the number of active (i.e., intersected) cells [ZN03]. Of the RMT operations listed in Section 2.2, Steps 2, 3, 4, and 6 are analogous to the processing in MC; the processing for these steps can be expected to be related to N and A_t. The first step is related only to N. The vertex clustering in Step 5 involves determining if a cluster should be formed and computing the cluster vertex location. This processing is greatly affected by the number of sample points to which the intersected edges are connected and the positions of the intersections on the edges. Thus, its computational effort could be related to factors such as A_e.

In this section, we describe a series of experiments that were performed to evaluate potential work estimation models that were based on combinations of the factors N, A_t, A_{vh}, A_e, and A_{vs}. For each set of factors, an optimal linear combination of factors was determined using least squares evaluation of a set of execution times for RMT-based isosurfacing, as described in Section 3. The reference isoqueries used for the experiment were 10 isoqueries on each of three medical datasets.

The goodness of the models was then evaluated on a conventional PC using 256 isoqueries on each of four medical imaging datasets different than those used to estimate the weights. (The parallel implementation uses only the most appropriate model, and weights for it were recomputed using the same isoqueries. Weights were determined based on runs in the parallel environment described in Section 5.) The models evaluated included four two-parameter models, four three-parameter models, and two four-parameter models. The two-parameter models were found to be highly inaccurate. The average relative errors in work estimation for these models for the isoqueries on one of the tested datasets, a CT dataset of a lobster, are shown in Table 1.

The three- and four-parameter models exhibited competitive estimations for the isoqueries. The average relative errors for the models for the lobster dataset isoqueries are shown in Tables 2 and 3. To save space, only the unique parameter IDs are listed in the three-parameter table (i.e., each model includes the factor N). Excepting the $T(N, A_e, A_t)$, $T(N, A_{vs}, A_t)$ and $T(N, A_e, A_t)$ estimates, each model was the most accurate for one set of isoqueries. Overall, the model (N, A_{vh}, A_e) was slightly more accurate than the other three-parameter models. On average over all the trials, the four-parameter models were the most accurate, however.

We also determined the amount of computational overhead associated with applying each model. Determining the value of A_t is relatively expensive, which makes the overhead associated with the four-parameter models unacceptably high. The model (N, A_{vh}, A_e) could be computed most quickly among the three-parameter models—on average, work estimation using it took 4.25% of the time for isosurface extraction; it is relatively accurate and has an overhead well-suited for parallel RMT. The other models ranged from an average of 6.8% to 25.4% overhead. The four-parameter models consistently required above 10% overhead, which

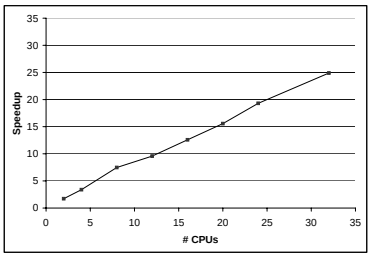


Figure 3: *Speedups for isosurface extractions from MRI Dataset*

(N, A_{vs}, A_t, A_e)	(N, A_{vh}, A_t, A_e)
17.6	14.1

Table 3: *Avg. Relative Error (%) in Workload Estimation Models, 4-parameter models (Lobster Dataset)*

makes them impractical. Similarly, a five-parameter model is even less practical and hence was not considered here.

5. Experimental Results and Analysis

In this section, the performance of our parallel RMT is reported. Peformance testing was done on the National Center for Supercomputing Applications (NCSA) linux cluster, which has more than 1000 nodes, each with two 3.2 GHz Intel Xeon CPUs. Each node has 3GB RAM and is interconnected to the other nodes via a Myrinet 2000 network.

An isosurface rendering for a computerized tomography dataset of a lobster (called Lobster) using our parallel RMT scheme is shown in Figure 2.

The isosurface extraction computation speedups for one isoquery are shown in Figure 3. The isoqueries were performed on an MRI dataset on the NCSA cluster. We observe that the performance of the isosurfacing exhibits a nearly linear speedup. However, for the datasets available to us, the RMT generally produced meshes with more facets than if MC isosurfacing was applied. This result conflicts with the published reports in [TPG99]. The discrepancy is partially accounted for by the parallel method's inability to form large clusters that span many blocklets. Since such clusters appear to be uncommon, more testing is needed to determine if the modalities of data available to us is less well-suited to formation of clusters than the data used in [TPG99].

6. Conclusion

A scheme for parallel RMT-based isosurface extraction has been described and comparatively evaluated. RMT is attractive for isosurfacing because it has the potential to yield a more compact isosurface description than does the popular MC. The scheme uses an accurate model of work estimation that can be computed rapidly, allowing good utilization of processing elements.

Acknowledgements

We appreciate the NCSA grant of time on the Xeon cluster. We also thank Ms. Jung-Im Choi for her assistance in preparation of this manuscript. The work described was partially supported by a grant from the National Science Foundation.

References

[AD03] ADAMS P., DOMMERMUTH D.: Visualization of steep breaking waves and thin spray sheets around a ship. In *Proc., Vis. '03* (2003), pp. 555–559.

[CFSW01] CHIANG Y.-J., FARIAS R., SILVA C., WEI B.: A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *Proc., IEEE 2001 Symp. on Parallel and Large-data Vis. and Graphics (PVG'01)* (2001), pp. 59–66.

[CP98] CHAN S. L., PURISIMA E. O.: A new tetrahedral tesselation scheme for isosurface generation. *Computers & Graphics 22*, 1 (1998), 83–90.

[GR00] GERSTNER T., RUMPF M.: Multiresolutional parallel isosurface extraction based on tetrahedral bisection. In *Volume Graphics* (London, 2000), ed. by M. Chen, A. Kaufman, and R. Yagel, Springer, pp. 267–278.

[LC87] LORENSEN W. E., CLINE H. E.: Marching Cubes: A high resolution 3D surface reconstruction algorithm. *Computer Graphics 21*, 4 (1987), 163–169.

[LT97] LOW K.-L., TAN T.-S.: Model simplification using vertex-clustering. In *Proc., 1997 Symp. on Interactive 3D Graphics* (1997), pp. 75–81.

[MN95] MIGUET S., NICOD J.-M.: A load-balanced parallel implementation of the Marching-Cubes algorithm. In *Proc., High Perf. Comp. Symp.'95* (1995), pp. 229–239.

[Nie03] NIELSON G. M.: On marching cubes. *IEEE Trans. on Vis. and Computer Graphics 9*, 3 (2003), 283–297.

[PT90] PAYNE B. A., TOGA A. W.: Surface mapping brain function on 3D models. *IEEE Computer Graphics and Applications 10*, 5 (1990), 33–41.

[SB00] SKALA V., BRUSI A.: Two methods for isosurface extraction from volumetric data and their comparison. *Machine Graphics & Vision 9* (2000), 149–166.

[TPG99] TREECE G., PRAGER R., GEE A.: Regularised marching tetrahedra: Improved iso-surface extraction. *Computers & Graphics 17* (1999), 397–414.

[ZN03] ZHANG H., NEWMAN T.: Efficient parallel out-of-core isosurface extraction. In *Proc., IEEE 2003 Symp. on Parallel and Large-data Vis. and Graphics (PVG'03)* (2003), pp. 9–16.

Eurographics Symposium on Parallel Graphics and Visualization - Short Papers (2006)
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

A Distributed Memory GPU Implementation of the Boris Particle Pusher Algorithm

P. Abreu[1], J.M. Pereira[2] and L.O. Silva[1]

[1]GoLP/Centro de Física dos Plasmas, Instituto Superior Técnico, Lisboa, Portugal
[2]IST/INESC-ID, Lisboa, Portugal

Abstract

The Boris pusher is a numerical algorithm to advance charged particles in an electromagnetic field. It is widely used in numerical simulations in Plasma Physics. This short paper explains the implementation of the Boris pusher algorithm on stream processors, in particular on a modern Graphics Processor Unit (GPU) with programmable shading capabilities, and explores the parallelization of the code on several GPUs.

Categories and Subject Descriptors (according to ACM CCS): J.2 [Computer Applications]: Physics; I.3.1 [Computer Graphics]: Parallel Processing; I.3.2 [Computer Graphics]: Distributed/Network Graphics .

1. Introduction

Numerical simulations in plasma physics [Daw83, BL91] often use an algorithm known as the Boris pusher [Bor70] to advance charge particles in the presence of an electromagnetic (EM) field. It is a second-order time centered numerical technique to solve the equations of motion

$$\begin{aligned} m\frac{d\mathbf{v}}{dt} &= q(\mathbf{E}+\mathbf{v}\times\mathbf{B}) \\ \frac{d\mathbf{r}}{dt} &= \mathbf{v} \end{aligned}$$

and it has been successful applied in many simulation algorithms [HE88]. In particular, it has been widely adapted for particle-in-cell (PIC) codes. The PIC codes belong to the general class of particle-mesh algorithms, widely used not only in plasma physics but also in fluid dynamics [HSW65]. A detailed description of a state-of-the-art PIC code can be found here [FST*02].

In a PIC code, physical quantities that are related to space (like the EM fields or the current) are stored as vertices of space cells, while quantities that are directly related with particles (like the charge or the momentum) are stored with the particles. When a cell quantity is needed at a particle position (e.g., when the EM fields needs to be determined in order to update a particle velocity or position), that value is interpolated at the particle position. To update the cell quantities, the particles' effect on that quantity has to be calculated. In PIC codes, the sources of the EM fields are the currents and the charges of the particles. The fields are determined from Maxwell's equations after depositing the current associated with each particle on the grid cells.

The particle push is one of the most time consuming steps in PIC codes. Since it uses vectorial quantities and it often needs to do vectorial operations, the Boris pusher is a very good candidate for high code acceleration, by adapting the algorithm to use vector and stream instructions (SIMD). Although most processors have vector instructions available, like Altivec on the PowerPC and MMX/SSE/SSE2/SSE3 on the Intel/AMD, they are usually working at the top of their capabilities during a numerical simulation. On the other hand, recent Graphics Processor Units (GPUs) have SIMD capabilities and have shown to provide good performance not only on streaming applications by performing the same operation over large collections of data as well as on applications that have sufficient parallelism and computational intensity to hide memory latency [BFH*04]. GPUs like the ATI X1900 series [ATI06] and the NVIDIA GeForce 7800 series [NVI05], feature both programmable vertex and fragment processors and provide support for floating point operations, making them available targets for streaming computation.

Taking advantage of these characteristics and the fact that GPUs are usually idle during numeric computations, in this short paper we describe the programming of a simplified PIC

code in the GPU's fragment processor as well as we discuss some issues of a distributed implementation over a cluster of GPUs.

2. The Boris pusher in a simplified PIC code

The computational steps of the Boris pusher are as follows:

1. Starting at a given time t, with a time step Δt,
2. the values of the velocity are given for $t - \frac{\Delta t}{2}$: $\mathbf{v}_{t-\frac{\Delta t}{2}}$.
3. However, the position of the particles $\mathbf{r}_t$ and the electromagnetic fields $\mathbf{E}_t$ and $\mathbf{B}_t$ are time centered at t.
4. Calculation of $\mathbf{v}_{t+\frac{\Delta t}{2}}$ with:

$$\begin{aligned}
\mathbf{v}^- &= \mathbf{v}_{t-\frac{\Delta t}{2}} + \frac{q}{m}\frac{\Delta t}{2}\mathbf{E}_t \\
\mathbf{v}' &= \mathbf{v}^- + \frac{q}{m}\frac{\Delta t}{2}(\mathbf{v}^- \times \mathbf{B}_t) \\
\mathbf{v}^+ &= \mathbf{v}^- + \frac{2\frac{q}{m}\frac{\Delta t}{2}}{1+\left(\frac{q}{m}\frac{\Delta t}{2}B_t\right)^2}(\mathbf{v}' \times \mathbf{B}_t) \\
\mathbf{v}_{t+\frac{\Delta t}{2}} &= \mathbf{v}^+ + \frac{q}{m}\frac{\Delta t}{2}\mathbf{E}_t
\end{aligned}$$

5. Calculation of $\mathbf{r}_{t+\Delta t}$ with:

$$\mathbf{r}_{t+\Delta t} = \mathbf{r}_t + \Delta t\, \mathbf{v}_{t+\frac{\Delta t}{2}}.$$

When used in a PIC code, $\mathbf{E}$ and $\mathbf{B}$ are defined at the each cell's vertices, and they have to be interpolated at $\mathbf{r}_t$.

A simplified PIC code, where the self-consistent evolution of the fields is neglected, can be resumed as follows:

1. Define the initial conditions:

$$t = t_0, \quad \mathbf{r}_{t_0}, \quad \mathbf{v}_{t_0-\frac{\Delta t}{2}}.$$

2. Calculate $\mathbf{B}_t$ and $\mathbf{E}_t$.
3. For each particle:
   a. Interpolate $\mathbf{B}_t$ and $\mathbf{E}_t$ at $\mathbf{r}_t$.
   b. Push the particles, getting the new positions, with the Boris pusher.
4. Advance the time by Δt and go to 2.

3. Streaming the Boris pusher

Modern GPU have been shown useful for many scientific computations [LLM04, GGHM05]. Recent work has shown the feasibility of using the stream capabilities of modern GPU for particle tracing [KKKW05], mainly due to the following reasons:

- Programmable shaders offer a programmable environment with highly accelerated vectorial operations.
- It is possible to allocate GPU memory that can be used as a texture map or a vertex array alternatively. This gives us the ability to run the whole algorithm with minimal data transfer between CPU and GPU memory.
- GPU support for float precision is improving. Recent processors can use full 32-bit floats in the entire pipeline, and 16-bit floats (OpenEXR format) for textures.

These capabilities of modern GPUs make them good hardware candidates for implementing the Boris pusher and the simplified PIC code explained in the previous section.

The implementation starts by allocating and initializing the textures that will hold the values of the vectorial quantities. The highest precision available for textures is 16-bit floats in the OpenEXR format.

For N particles, $\mathbf{r}$ and $\mathbf{v}$ are stored in 1D RGB textures of length N (or of the next power of two, since non power of two textures can still be penalized in performance). Two textures for each one of these quantities are needed, one that stores $\mathbf{r}_t$ and $\mathbf{v}_{t-\frac{\Delta t}{2}}$, and another for $\mathbf{r}_{t+\Delta t}$ and $\mathbf{v}_{t+\frac{\Delta t}{2}}$. We call this textures T_{r0}, T_{v0}, T_{r1} and T_{v1}. Two 3D RGB textures, T_E and T_B, are also needed, one for $\mathbf{E}$ and the other for $\mathbf{B}$. They have the same dimension has the number of cells of the simulation space (again, some power of two ajustment might be needed). Figure 1 illustrates how this textures are used in the algorithm.

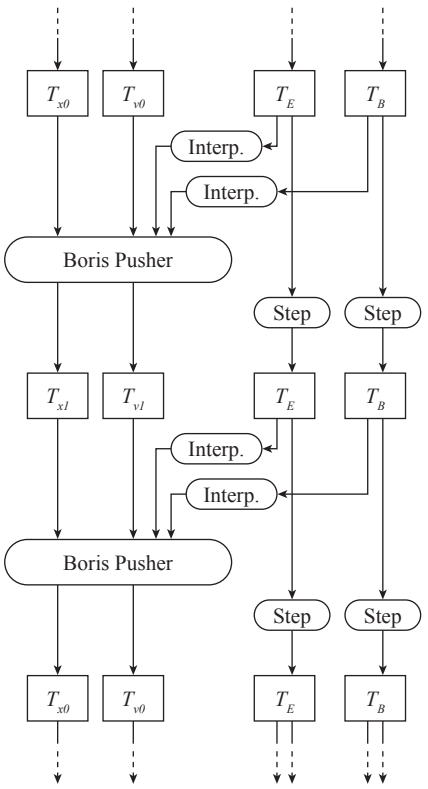


Figure 1: *The use of textures in the GPU implementation of the Boris pusher.*

Scalar quantities, like Δt, m and q, are passed as uniform shader parameters.

For each particle, **E** and **B** are interpolated at the particle position, after it is read from T_{r0}. The new positions are calculated using the Boris pusher: $\mathbf{v}_{t+\frac{\Delta t}{2}}$ is calculated and rendered to T_{v1}, and $\mathbf{r}_{t+\Delta t}$ is calculated and rendered to the T_{r1} texture. At this step, T_{r1} can also be rendered to the screen.

The time is advanced and a new cycle begins. T_E and T_B are updated, and T_{r1} and T_{v1} are used to calculate the new **v** and **r**, that are rendered to T_{v0} and T_{r0}.

Boundary conditions have to be implemented. In our simulation, we have use a periodic boundary, so that the particles are re-injected when they leave the simulation box.

4. Parallelization Issues

With this implementation, we are able to have a real-time display of a simple PIC algorithm with a Boris pusher. However, the memory size of modern GPUs is still an important limitation. A small simulation, with 128^3 cells and 8 particles per cell requires 408 MB just for texture memory, which might not be available even on video cards with 512 MB RAM.

One way to overcome this limitation is to distribute the simulation over several GPUs. Similar approaches have been done with other algorithms [KKKW05, HHH05]. With our simplified PIC code, the particles do not influence the fields. The Boris pusher can be parallelized with very little penalty if we only partition the particles over each node, but still use the full fields. In our previous example, memory requirements are reduced from 408 MB to $24 + 384/P$, where P is the number of available GPU processors. With just 2 GPUs, this results in 216 MB of texture memory, which fits well in processors with 512 MB video memory.

More interesting is the possibility to run simulations of a reasonable size, like 256^3 cells and 8 particles per cell. This is a $8\times$ increase on texture memory requirements. On a 16 node cluster, 288 MB are needed per node. As long as communication between the nodes is kept to a minimum, this is an acceptable solution.

5. Cluster implementation

An implementation of the techniques described in this short paper have been implemented on commodity hardware that is normally used as part of a Grid node for plasma physics simulation. Each working node (WN) was configured with an AMD Athlon 64 3200+ CPU with 1 GB RAM and a nVidia 6600GT PCIe video card with 128 MB RAM. An user interface node (UI) was also used, which was responsible for initializing the textures with the data and for launching the GPU processes on the working nodes. The LAM MPI implementation was used as the communication interface between the nodes [GLS99].

The application running on the UI detected the WN available and distributed the particles. After receiving the particles, each WN ran the simulation for a certain simulation time or number of steps (set by the UI). During this time, no node communication was needed. The final result (particle position and velocity) was sent from the WN to the UI, where it was stored and displayed.

6. Performance

Simulations run on a system with one UI and two WNs showed a very good performance, comparable to the simulation on the CPU. Overhead caused by the transfer of textures between CPU and GPU memory was compensated by the SIMD characteristics of the GPU. Even a full PIC code using the GPU implementation of the Boris pusher showed promising results. An improved system is being built, with more nodes and better video cards (more video memory and improved pipelines), so that more realistic simulations can be run.

7. Limitations and future work

The biggest limitation in this GPU implementation of the Boris pusher is the lack of support for double precision floats in the GPU. To study some very detailed plasma behavior, half or single precision is not enough. However, even in these cases a GPU Boris pusher with limited float precision is still helpful as a way to see a real-time evolution of the simulation, that helps to quickly grasp the general behavior of the particles.

The limited video memory available to most GPUs (usually less than 1 GB) is also a constrain on the application of this algorithm on big-scale simulations. Parallelization is a good option for a simplified PIC code, since very little node communication is required.

However, using this algorithm with a full PIC code is possible and an interesting challenge. In full PIC codes, the moving charges creates a current **J** which influences the EM fields. Two steps have to be introduced after the Boris pusher in the simplified PIC algorithm detailed in section 2:

- calculate the new current **J** from the position of the charges and from the momenta, called current depositing;
- solve the field advance equations for the **E** and **B** fields, so that the influence of **J** is taken into account.

This not only increases memory requirements (another texture has to be available to read from and to render to), but also forces node communication at each time step, since in order to deposit the current, particles stored in different processors might be needed.

The usual approach is to pass particles from one processor to another, as they leave a cell stored in one processor and enter another cell, stored in a different processor. This would

mean creating and deleting particles from textures, which is very computational expensive for the GPU. However, if video memory allows to store textures for the full **E**, **B** and **J**, then particles might not have to change textures from one GPU to another. It could be enough to keep track of which particle from which node is influencing **J** for depositing. A possible way to implement this could be by using a RGBA texture and storing this information in the alpha channel. This would further increase texture memory requirements, but would minimize both the transfer of data between the GPU and the CPU, and the node communication per cycle.

8. Acknowledgments

This work was partially supported by FCT, Portugal, SFRH/BD/17870/2004. The authors would like to thank Ricardo Fonseca, Luís Gargaté and Michael Marti from GoLP, IST, for their help on some fine details of the Boris Pusher algorithm.

References

[ATI06] ATI: Radeon x1900 product site, 2006. http://www.ati.com/products/RadeonX1900.

[BFH*04] BUCK I., FOLEY T., HORN D., SUGERMAN J., HANRAHAN P., HOUSTON M., FATAHALIAN K.: Brook for gpus: Stream computing on graphics hardware. In *Proceedings of the ACM SIGGRAPH 2004* (2004).

[BL91] BIRDSALL C. K., LANGDON A. B.: *Plasma Physics Via Computer Simulation.* Adam Hilger, UK, 1991.

[Bor70] BORIS J. P.: Relativistic plasma simulation—optimization of a hybrid code. In *Proc. Fourth Conference on the Numerical Simulation of Plasmas* (Nov. 1970), pp. 3–67.

[Daw83] DAWON J. M.: Particle simulation of plasmas. *Rev. Mod. Phys. 55*, 2 (Apr. 1983), 403–447.

[FST*02] FONSECA R. A., SILVA L. O., TSUNG F. S., DECYK V. K., LU W., REN C., MORI W., DENG S., LEE S., KATSOULEAS T., ADAM J.: Osiris: A three-dimensional, fully relativistic particle in cell code for modeling plasma based accelerators. In *Lect. Notes in Comp. Sci.* (2002), vol. 2331, Springer-Verlag, Heidelberg, pp. 342–ff.

[GGHM05] GALOPPO N., GOVINDARAJU N. K., HENSON M., MANOCHA D.: Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. (2005 ACM/IEEE Conference on Supercomputing).

[GLS99] GROPP W., LUSK E., SKJELLUM A.: *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[HE88] HOCKNEY R. W., EASTWOOD J. W.: *Computer Simulation Using Particles.* Institute of Physics Publishing, Bristol and Philadelphia, 1988.

[HHH05] HORN D. R., HOUSTON M., HANRAHAN P.: Clawhmmer: A streaming hmmer-search implementation. (2005 ACM/IEEE Conference on Supercomputing).

[HSW65] HARLOW F. H., SHANNON J. P., WELCH J. E.: Liquid waves by computer. *Science 149*, 1092-3 (1965).

[KKKW05] KRÜGER J., KIPFER P., KONDRATIEVA P., WESTERMANN R.: A particle system for interactive visualization of 3d flows. *IEEE Trans. Vis. Comput. Graphics 11*, 6 (Nov. 2005), 744–756.

[LLM04] LASTRA A., LIN M., MANOCHA D.: Visualization in grid computing environments. 155–162. (ACM Workshop on General Purpose Computation on Graphics Processors).

[NVI05] NVIDIA: Geforce 7800 product site, 2005. http://www.nvidia.com/page/geforce_7800.

Eurographics Symposium on Parallel Graphics and Visualization - Short Papers (2006)
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

3-D Critical Points Computed by Nested OpenMP

Andreas Gerndt[1], Samuel Sarholz[2], Marc Wolter[1], Dieter an Mey[2], Torsten Kuhlen[1], Christian Bischof[2]

[1]Virtual Reality Group, RWTH Aachen University, Germany
[2]Center for Computing and Communication, RWTH Aachen University, Germany

Abstract

Extraction of complex data structures like vector field topologies in large-scale, unsteady flow field data sets for the interactive exploration in virtual environments cannot be carried out without parallelization strategies. We present an approach based on Nested OpenMP to find critical points, which are the essential parts of velocity field topologies. We evaluate our parallelization scheme on several multi-block data sets and present the results for dynamically assigned threads to the different parallelization levels. Our experience suggests that upcoming massively multi-threaded processor architectures can be very advantageously for large-scale feature extractions.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Parallel processing; D.4.1 [Operation Systems]: Multiprocessing/multiprogramming/multitasking; G.1.0 [Numerical Analysis]: Parallel algorithms; I.3.7 [Computer Graphics]: Virtual reality

1. Introduction

Helman and Hesselink [HH91] were one of the first who described approaches to visualize the topology of flow fields computed by Computational Fluid Dynamics (CFD). The most important step is to find all locations where the velocities vanish. These positions are also called critical points. Once detected, they can be classified, and separation lines may be computed. In the meantime, a variety of improvements were presented by several authors. A brief overview of common algorithms for vector field topology extraction is given by [TWHS03]

Our implemented algorithm to find first-order critical points in structured grids is based on the bisection scheme introduced by Globus et al. [GLL91]. In general, linearity is assumed between cell nodes so that tri-linear interpolation can be utilized in order to determine the velocity at an arbitrary position within a hexahedron. For curvilinear grids, as a first step, we have to transform each curved hexahedral cell from physical space ($\mathcal{P}$-space) into computational space ($\mathcal{C}$-space). Thereafter, the Newthon-Raphson iteration is usually applied to find the exact zero velocity location within such a cube.

This approach would always find only one root although multiple roots within a cell are possible. Therefore, before applying Newton-Raphson iteration, a sub-division scheme called bisection is carried out dividing a hexahedron into eight sub-cells. But not each hexahedral cell contains critical points. A heuristic compares the signs of the x-, y-, and z-components of all velocity vectors stored at the cell vertices in order to estimate whether a cell might include roots of the tri-linear function or not. These cells are called candidate cells and are exclusively considered for the subsequent bisection scheme. Other cells are discarded.

After one bisection step, a candidate cell test is again carried out. Sub-cells determined as candidates are used for further bisection steps. This iterative sub-division can now be applied until a certain bisection depth is reached. After the bisection terminates, the actual critical point position can now be determined for all remaining sub-cells by the Newton-Raphson iteration.

Critical points classify the flow field behavior in their immediate vicinity. For the assessment, the velocity gradient tensor J_v at that position is needed. The eigensystem of J_v consists of three real eigenvalues and eigenvectors, or two complex conjugate and one real eigenvalues and eigenvectors, respectively. Exclusively real values identify nodes, otherwise spirals are detected [TWHS03]. Streamlines along real eigenvectors are called separatrices and give an impression of the topology of the velocity field (cf. figure 1). The parallel computation of separatrices, however, is not covered by this paper.

Figure 1: *Engine, critical point symbols and separatrices.*

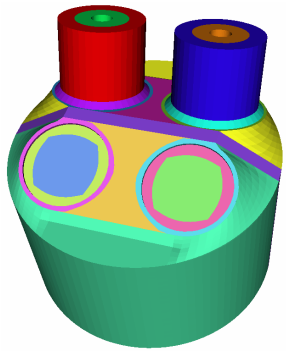


Figure 2: *Differentially colored blocks of the Engine MB-data set.*

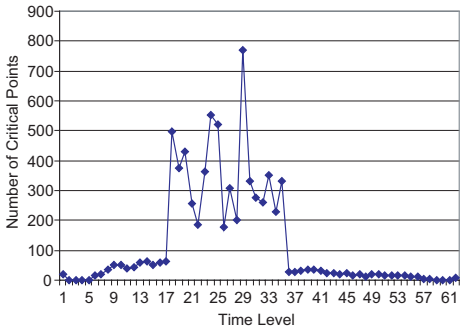


Figure 3: *Engine, amount of critical points.*

Large-scale flow simulations investigate, in general, unsteady phenomena. Additionally, complex flow fields are frequently decomposed into several grids. In the case of critical point computation, the used algorithm is a cell-based approach, i.e. no additional information of neighboring cells are needed. Moreover, the bisection scheme works on subcells which can also be processed locally. Therefore, the hypothesis of the work presented by this paper is that we might achieve significant speed-up by the use of parallelization on all depicted levels: time level, block level, cell level, and subcell level.

The remaining paper is structured as follows: In section 2, data sets used to evaluate the critical point extraction are examined. Thereafter, we present strategies for nested parallelization by means of OpenMP. Finally, experimental results are presented and evaluated in section 4.

2. Data Sets

A variety of data sets were utilized in order to explore the behavior of the critical point computation algorithm. Two are selected for this short paper. The smallest but most interesting data set is a spark ignition engine where only the inflow and compression phase were simulated. It is a multi-block (MB) data set where the chamber and the inlet valves are decomposed into 23 structured grids (cf. figure 2). The position and the number of vertices and cells may change between two successive time levels.

For all data sets, the total amount of occurring critical points were computed. In figure 3 the found critical points of the engine data set are presented per time level.

One can see that critical points are primarily detected during the inflow process (time level 1 to 34). To identify the cost of the bisection approach, the runtime for the basic load (cell traversal, $\mathcal{C}$-space transformation and initial candidate cell test) as well as the total runtime were measured. The results for the engine are depicted in figure 4. Whereas the initial load is independent of the selected bisection depth, the total runtime is not. The presented result was measured using a bisection depth of 6.

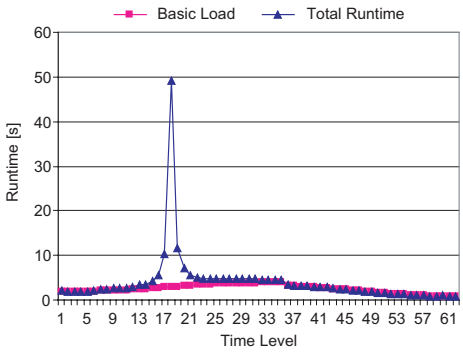


Figure 4: *Engine, basic load and total computation load.*

A high peak can be detected around time level 18. This is the situation where the valves stop and change their directions. Thereafter, they move against the inflow. For that situation, velocities close to the valve boundary are almost

zero. Here, the bisection approach detects a large amount of sub-cells with opposite directions which results in high bisection expense. The Newton-Raphson iteration needs also a lot of time as it does not converge quickly or not at all.

The second data set describes a shock simulation. Instead of the other investigated data sets, this one is not a multi-block data set but a rectilinear one. It contains more time levels, and the number of critical points increases over time. Because of dominant flow direction and the rectangular shape of the cells, the bisection approach finds critical points accurately and fast (cf. figure 5).

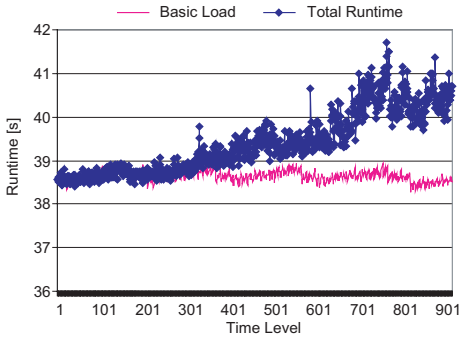


Figure 5: *Shock, basic and total load.*

Due to their various features, we thus consider these data sets good candidates for evaluating the parallel strategy.

3. Parallelization

The following measurements were carried out on a Sun Fire E25K equipped with 72 UltraSparc IV dual-core processors. As we are just interested in measuring the runtime of the parallel feature extraction, we always preloaded the entire data set into the main memory.

OpenMP [CMD*98] was designed for a straightforward parallelization of sequential codes for shared-memory systems by means of `pragma` directives. The OpenMP parallel loop directive currently offers three different schedule kinds, which permits careful control of the distribution of work to the threads. The simplest loop schedule is `static`, which causes an equal distribution of loop iterations to all threads. On the other hand, OpenMP offers the `dynamic` schedule, which assigns iterations to the threads after they are done with their previous work. This eliminates any load imbalances as far as possible, but it also involves a higher overhead. The third schedule kind is the `guided` schedule, which dynamically distributes the work to the threads but also reduces the overhead.

All three schedule kinds offer a chunk size parameter, which allows the bundling of iterations into chunks of the specified size as they are assigned to the threads. This also reduces the overhead of the scheduling mechanism. Whereas the `static` and the `dynamic` schedules always assign chunks of the given size – except for the last chunk of a loop, which may have fewer iterations – the `guided` schedule starts with larger chunks and then gradually reduces their size towards the end of the loop iteration space. The next chunk size is proportional to the number of unassigned iterations divided by the number of threads and is calculated with the equation (1). Sun's OpenMP implementation allows adjusting the weight parameter c and uses 2 as a default value. A large weight parameter leads to a schedule which is similar to dynamic. A smaller c reduces the overhead but might increase the load imbalance.

$$chunk\_size = \#unassigned\_iterations/(c \cdot \#threads) \quad (1)$$

Sun's OpenMP implementation also supports nested parallelization. In our algorithm, we find three loops processing time steps, blocks, and cells, which are well suited for parallelization. The varying parameter in our measurements, above all, were the number of threads assigned to each of these levels. The `dynamic` schedule seemed to be an adequate choice for higher parallelization levels. For cell level, however, we expected that `guided` leads to a optimum between scheduling overhead and load balancing.

4. Results

Let n be the number of threads involved. We used up to 128 processors and measured all powers of two combinations of thread distributions to the parallelization levels. The amount of threads for the time level is denoted as ti and for block level as bj. The remaining $n - i \cdot j$ threads are assigned to the cell level (denoted as ck). For all but the engine data set, we selected a maximal bisection depth of 10. The engine is only divided up to a depth of 6. The speed-up results obtained by the nested parallel critical point computation utilizing the engine data set are depicted in figure 6.

Cell based parallelization is rather inefficient. In fact, the speed-up drops again using more than 32 processors. Moving threads to higher levels improves the speed-up and reaches a maximum with 4 time level threads and 4 block level threads. Using even more threads on higher levels is not profitable.

The speed-up is clearly improved for the shock data set (cf. figure 7). The [t32,b1] measurement shows an almost optimum result. As this set only contains a single block, only two levels of parallelism are available.

4.1. Variation of Loop Schedules

In order to assess the measured results, we also modified scheduling parameters and chunk sizes. For instance, if only `static` is used to assign data to OpenMP threads on all

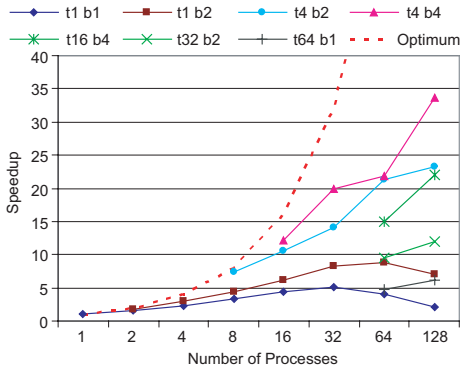


Figure 6: *Engine, speed-up of selected probes.*

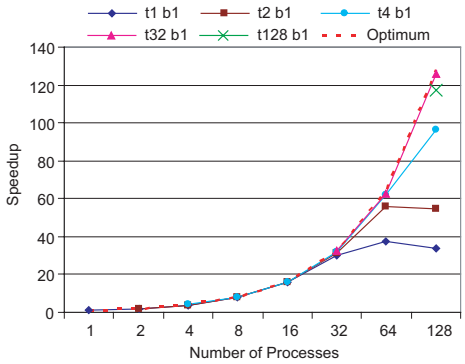


Figure 7: *Shock, speed-up of selected probes.*

three levels, the achieved speed-ups drop to 30% (engine: 10.38) and 92% (shock: 116.19), respectively.

The engine data set suffers from the heavy load imbalance introduced by time level 18. To improve the speed-up for this case, we also experimented with parallelization on sub-cell level. If 8 threads were used on sub-cell level to process the mainly affected blocks 20 and 21, the runtime could be reduced only by a factor of 2.36.

Alternatively, we also modified the guided weight parameter (cf. equation (1)) which yielded the results depicted in figure 8. A weight of 20 shows the best speed-up when using 8 threads to process block 20 and 21. This modification led to an improvement of the speed-up from 4.16 to 6.37. Furthermore, applying the weight factor 20 for the entire engine data set, the overall speed-up using 128 processors ([t4,b4,c8]) could be improved from 33.67 (cf. figure 6) to 55.18!

5. Conclusions and Future Work

Multi-level parallelization with OpenMP paid off for all investigated data set. The elapsed time of the critical point computation for the shock data set could be reduced from

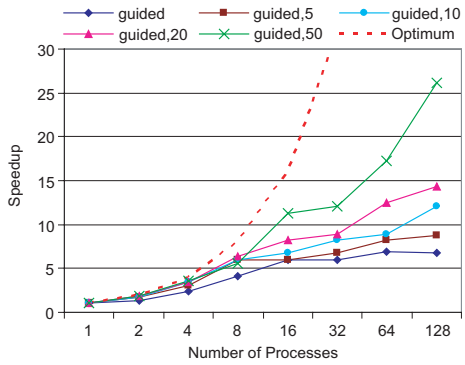


Figure 8: *Engine, varying weights for* `guided` *processing block 20 and 21 at time level 18.*

12 hours and 18 minutes down to about 6 minutes, a tremendous reduction in runtime.

Nested parallelization offers several degrees of liberty in choosing the number of threads and the loop schedule on each level. Therefore, we are currently working on strategies to automatically adjust the available parameters in order to achieve an optimized speed-up. One promising approach might be the usage of the task queuing work-sharing construct as it is currently implemented in the Intel compiler. A similar mechanism can be expected to be accepted for the upcoming OpenMP specification.

6. Acknowledge

We would like to thank the German Research Foundation (DFG), who funded some of the methodical work. We are also grateful to the Institute of Aerodynamics, RWTH Aachen University, who kindly made available data sets for evaluation purposes.

References

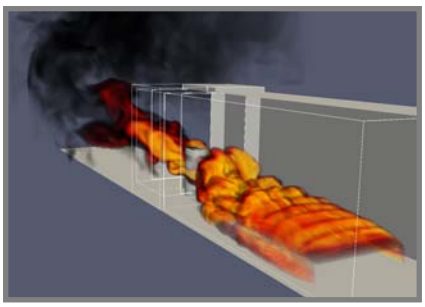
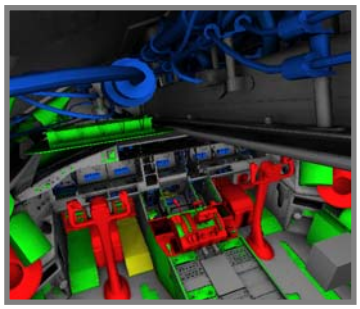
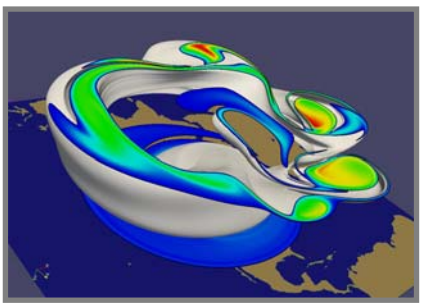
[CMD*98] CHANDRA R., MENON R., DAGUM L., KOHR D., MAYDAN D., MCDONALD J.: *Parallel Programming in OpenMP*. Morgan Kaufmann, 1998.

[GLL91] GLOBUS A., LEVIT C., LASINKI T.: A tool for visualization the topology of three-dimensional vector fields. In *Proceedings, IEEE Visualization, San Jose, CA* (1991), pp. 33–40.

[HH91] HELMAN J. L., HESSELINK L.: Visualizing vector field topology in fluid flows. *IEEE Computer Graphics and Applications 11*, 3 (1991), 36–46.

[TWHS03] THEISEL H., WEINKAUF T., HEGE H.-C., SEIDEL H.-P.: Saddle connectors - an approach to visualizing the topological skeleton of complex 3d vector fields. In *Proceedings of IEEE Visualization, Seattle, WA* (2003), pp. 225–232.

This book contains the short papers presented at the 6th Eurographics Symposium on Parallel Graphics and Visualization, organized by the Eurographics Association in co-operation with ACM SIGGRAPH, which took place from May 11 – 12, 2006 in Braga, Portugal.

ISBN: 978-972-99166-3-2