Eurographics Symposium on Parallel Graphics and Visualization (2017)
J. C. Bennett, A. Telea (Editors)

A Task-Based Parallel Rendering Component For Large-Scale Visualization Applications

T. Biedert[1†], K. Werner[1] B. Hentschel[2] and C. Garth[1]

[1]University of Kaiserslautern, Germany
[2]RWTH Aachen University, Germany

Abstract

An increasingly heterogeneous system landscape in modern high performance computing requires the efficient and portable adaption of performant algorithms to diverse architectures. However, classic hybrid shared-memory/distributed systems are designed and tuned towards specific platforms, thus impeding development, usage and optimization of these approaches with respect to portability. We demonstrate a flexible parallel rendering framework built upon a task-based dynamic runtime environment enabling adaptable performance-oriented deployment on various platform configurations. Our task definition represents an effective and easy-to-control trade-off between sort-first and sort-last image compositing, enabling good scalability in combination with inherent dynamic load balancing. We conduct comprehensive benchmarks to verify the characteristics and potential of our novel task-based system design for high-performance visualization.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics

1. Introduction

High-fidelity computational simulation models have assumed a significant role in scientific research and engineering applications, thereby necessitating efficient visualization techniques at large scale. In recent years, parallel algorithms for concrete classes of visualization problems have been presented, such as direct volume rendering [HBC12] or integral curve computation [PCG*09]. Most large data approaches typically utilize a distributed memory model, where bulk-synchronous execution and communication using the Message Passing Interface (MPI) is standard. For improved scalability, hybrid approaches commonly resort to MPI for the coarse distribution of parallelly executable parts of an algorithm to a set of processes, where within each process a second concept - e.g. OpenMP, OpenCL, or CUDA - is used for additional finegrained parallelization of these steps.

These practices require detailed knowledge of the different parallelization concepts and often result in specific optimizations for certain platform configurations or obligating the usage of distinct hardware components, which complicates or even hinders portability towards other architectures. An additional challenge in the parallelization of visualization concepts is posed by the fact that, in contrast to simulation computations, visualization tasks are frequently bandwidth-limited and inherently unbalanced. Thus, achieving scalable parallel execution demands not only an efficient utilization of the available memory bandwidth, where memory accesses ideally are overlapping with computational tasks, but also dynamic load balancing.

Considering the general parallelization of complex algorithms against this background, in recent years the paradigm of task-based parallelization has been established [DG15]. Here, an algorithm is formulated as a set of tasks which can be carried out concurrently, where a single task represents an atomically executable subsequence of the algorithm. Interdependencies between tasks can be modelled explictly by the developer. These relationships are used by the underlying runtime environment to coordinate the parallel execution. Thus, with the help of the task graph the developer specifies *what* should be executed, whereas the *how* of the execution is left to the runtime environment [KHAL*14].

The task-based paradigm entails several crucial advantages. Conceptionally, tasks enable a more straightforwad formulation of massively parallel programs, where the maximum degree of parallelism is determined by the maximum width of the task graph for the given computation. Technically, the coordinated execution by the runtime environment ensures a flexible and transparent portability to diverse hardware platforms. Furthermore, task-based systems can inherently handle the parallel execution of dynamically changing computational loads by the principle of work stealing [DLS*09].

† *biedert@cs.uni-kl.de*

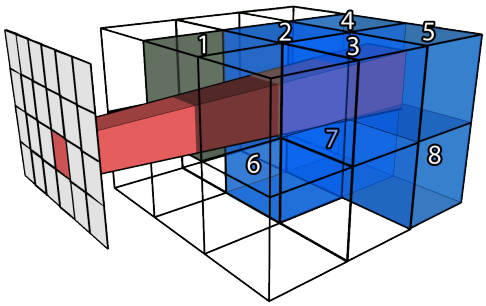


Figure 1: *Hybrid parallelization in both image and data space. For each tile, the set of visible blocks is topologically sorted and composited using a distributed binary tree communication scheme. The numbering represents an arbitrary enumeration of the visible blocks within the highlighted tile's frustum. See Figure 2 for the corresponding compositing tree. Note that block 1 is discarded due to empty-block skipping.*

In this context, the intent of this paper is to study a task-based system design for distributed direct volume rendering. Our task definition is based on hybrid parallelization in both image and data space (see Figure 1), thus representing an effective and easy-to-control trade-off between sort-first and sort-last image compositing. The presented asynchronous binary tree compositing scheme enables good scalability in combination with inherent dynamic load balancing.

The overall intent of this paper is to investigate possible advantages of such an approach for the design of large scale visualization systems. Specifically, after a brief review of relevant prior work (Section 2), we make the following contributions:

- In Sections 3 through 4, we describe a task-based formulation for a distributed direct volume rendering system.
- We conduct comprehensive benchmarks to verify the characteristics and potential of our novel task-based system design for high-performance visualization and describe results and analysis in Section 5.
- We anticipate that many enhancements and improvements are possible, and discuss a number of such opportunities in Section 6.

Our contribution is intended as a baseline demonstration of the applicability of the emerging task-based paradigm in large scale high performance computing to distributed algorithms and challenges in scientific visualization.

2. Related Work

2.1. Distributed Volume Rendering

Direct volume rendering [DCH88] represents a crucial class of algorithms used in scientific scalar field visualization. Today, direct volume rendering typically follows the principle of ray casting [Lev90], where primary rays are traced through a volumetric data set starting at a virtual camera and depending on the underlying sample locations optical properties are determined and accumulated along each ray.

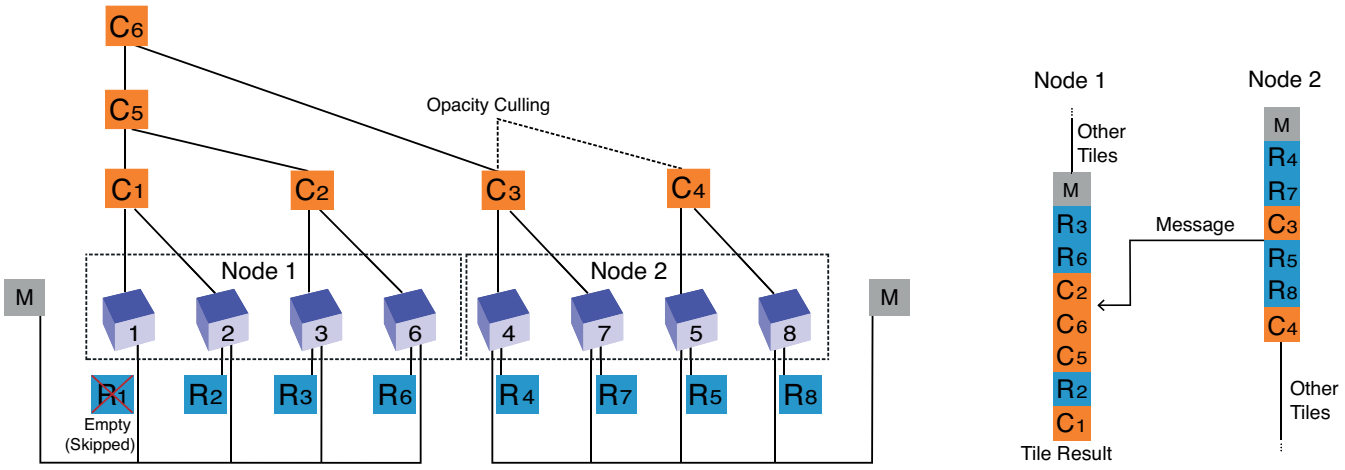
Following the nomenclature of [MCEF94], there are two fundamental approaches for the parallelization of volume ray casting: sort-first and sort-last volume rendering. In sort-first, the image plane is subdivided into rectangular tiles for which rendering is performed concurrently [CM93, BHPB03, MAWM11]. Sort-last algorithms perform parallelization based on a spatially disjunct partion of the input data, where each process computes a partial image of its assigned data. The resulting images are afterwards composed to the final output image. While both techniques have specific advantages and drawbacks [MAWM11], most scalable systems typically employ the sort-last approach, primarily due to the slower increase in image resolution compared to data size. Müller et al. present a hardware-accelerated sort-last approach, using block subdivision for fast empty-space skipping and performing dynamic load balancing by block redistribution based on previous computation times [MSE06]. Marchesin et al. achieve load balancing by dynamic restructuring of the underlying k-d tree [MMD06]. Navratil et al. use queue-based dynamic scheduling in order to increase ray coherence and memory bandwidth utilization, leading to improved L2-cache access patterns [NFLM07]. Childs et al. present a hybrid scheme, using distributed parallelization both in input data and in image space [CDM06].

Dedicated graphics and accelerator cards providing numerous processing cores have proven to be a powerful tool for computationally expensive applications such as ray casting on large data [BHP15, KWN*14]. Consequently, the prevalent usage of multi-/many-core processor architectures and accelerator cards in distributed high performance systems has given rise to diverse hybrid parallelization approaches. Peterka et al. implement hybrid parallel volume visualization of massive data sets on the IBM BlueGene/P architecture using MPI and POSIX-Threads, where up to 90% of the total runtime are dedicated to I/O [PYRM08]. Fogal et al. study direct volume visualization using OpenGL-based slicing on distributed memory multi-GPU clusters in combination with subsequent MPI-based compositing [FCS*10]. Howison et al. compare different common hybrid approaches based on POSIX-Threads, OpenMP and CUDA in combination with MPI for direct volume visualization, In general, hybrid techniques offer improved performance with reduced memory and communication overhead [HBC12].

A crucial bottleneck in the performance of massively parellel sort-last volume rendering algorthms is the final composition of the partial per-process images. A comparison of the common approaches (Direct Send [EP07, SML*03], Binary Swap [MPHK94], Radix-k [YWM08, PGR*09, KPH*10]) shows that notable runtime benefits can be achieved using hybrid strategies with variable granularity.

2.2. Task-Based Parallelization

Faced by the emergence of increasingly hierarchical and heterogeneous system architectures, the hybrid MPI-threading model prevalent in high performance computing turns out to be more and more suboptimal. The resulting parallelism is fragile due to the lack of a strict separation between computational kernel and parallel execution, in addition to the strong coupling with the underlying architectures.



(a) *Communication tree* **(b)** *Scheduling timeline*

Figure 2: *Communication pattern and possible task execution order for the eight blocks (numbered cubes) shown in Figure 1 after topological sorting (from left to right) distributed across two nodes. In this example, for each block a rendering task (R) is scheduled by the respective node, except for the first block which is empty. Additionally, each node calculates the necessary compositing steps and supplies each local block with the resulting meta information (M) it needs for sending to its receiving block. According to this meta information the ready blocks may send their image data to their receiving blocks, which will initiate the scheduling of composition tasks (C). In this example, the composited image of the partial images from block 4 and 7 is already opaque, allowing early initiation of the next compositing step, thereyby skipping the wait for the result of C4. The execution timeline assumes only a single local thread per node for simplicity. Due to the asynchronous execution of tasks the results of C2 and C3 are composited before C1 is performed. Note, that the execution of tasks may be interleaved with tasks from other tiles.*

In scientific high performance computing task-based dynamic runtime environments are considered as a promising alternative model [DG15], whose benefits have already been demonstrated in diverse disciplines. Haidar et al. present the dynamic scheduling of algorithms in linear algebra [HLYD11]. A dynamic runtime environment for grid workflows can be found in [AA07]. Notz et al. demonstrate a graph-based system design with a dynamic runtime environment for multiphysics software based on partial differential equations [NPS12]. The simulation of large biomolecular systems is shown in [KBM*08]. However, the majority of scientific applications has not yet integrated dynamic runtime environments, or is still in early experimental stages [DG15].

Current programming languages, libraries or runtime environments start to offer task-based programming models. A comparison of numerous independent runtime environments and task-based execution models can be found in [GABS13]. Popular frameworks for single shared memory multicore systems are the task implementation in the OpenMP standard (starting with version 3.0), the Intel Threading Building Blocks (TBB) library or Intel Cilk Plus. Our work heavily utilizes the HPX (High Performance ParalleX) framework [KHAL*14], which implements the ParalleX execution model and provides task-based parallelization across node boundaries. HPX manages an active global address space and focuses on latency hiding by the dynamic scheduling and asynchronous execution of fine-grained tasks with minimal context switching overhead. Other recent frameworks for distributed task-based parallelization include Charm++ [AGJ*14] and Legion [BTSA12], which has been used to explore the applicability of asynchronous many-task (AMT) programming models in the context of in-situ data analysis [PBH*16].

3. System Design

3.1. Task Granularity

In a task-parallel system the achievable degree of parallelization and scalability is crucially characterized by the so-called task granularity, i.e., the size of the individual tasks, balancing the width of the task graph versus individual task overhead.

Our approach aims to provide a flexibly tunable task granularity by subdividing in both image and data space, thus yielding a hybrid scheme between sort-first and sort-last compositing. Volume data is split into regular blocks of equal size, which are distributed across nodes, whereas the image plane is divided into rectangular tiles. The general approach is to render the visible blocks within each tile, compose all images per tile in correct order and eventually align all completed tile images to form the resulting output image.

An example of the hybrid subdivision in both tiles and blocks is illustrated in Figure 1. The corresponding communication tree for image compositing is depicted in Figure 2, also showing the interleaved scheduling order of the individual rendering and compositing tasks.

This scheme allows to balance the number of mutual partners for image compositing by tweaking both block and tile size, while also providing means to incorporate common optimization techniques such as empty-block skipping and early ray termination.

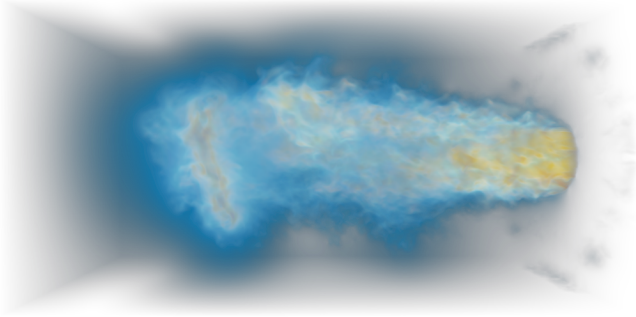


Figure 3: *The* jet *reference data set used for all benchmarks with* 2048^3 *voxels (32 GB) on a single node. For weak scaling, the number of voxels is increased proportionally to the number of cores, up to* 6502^3 *(275 gigavoxels) on 32 nodes. All renderings are performed at 3840x2160 resolution.*

The following Section 3.2 describes the interplay and dependencies of the individual rendering and compositing tasks in our novel distributed compositing scheme. The incorporation of additional optimizations is presented in Section 3.3.

3.2. Distributed Compositing

Initially, each node computes for all tiles the visible blocks within the respective viewing frustums and schedules for each local block a rendering task. As these render tasks begin execution, each node calculates a sequence of compositing steps for each tile. A compositing step consists of an initiating block, whose rendered image will be blended behind the image of the receiving block. The goal is to find a sequence of compositing steps so that every step maintains the correct z-ordering of fragments and ultimately after the last step one block holds the complete image for the respective tile. The node containing that last block will contribute the tile to the final output image.

First, all (including remote) blocks inside the viewing frustum of the tile are topologically sorted based on the Manhattan distance to the camera, so that each block in the sorting can never overlap a block prior to it.

A naïve approach would be to iterate over this sorting from back to front and schedule a compositing step for each block as initiator and its successor in the sorting as receiving block, thus yielding a correct, sequential compositing of all block images. This would result in every block being the initiator and receiver of a composition step exactly once, except for the frontmost block who never initiates a compositing and the backmost block who never receives one.

For a scheduled compositing step to start and execute the following dependencies are necessary: First, the rendering task for the initiating block needs to be finished. Second, the compositing step which will be received by the current initiator needs to be completed. An exception to this is the backmost block who can initiate compositing immediately after rendering.

Our communication pattern is more sophisticated than the aforementioned naïve sequential approach. Compositing steps are chosen to form a binary tree over the topological sorting of the involved blocks across node boundaries. Every second block initiates just after rendering is completed for the tile, with its successor in the sorting being the receiver. Every second of these receivers is scheduled to initiate a compositing step with its successive receiver and so on. In the end, every block was initiator exactly once, half of the blocks received once, a fourth received twice and so on. The number of times the frontmost block is receiver of a compositing step is equal to the height of the binary compositing tree formed by the compositing steps.

Note that while the compositing steps are determined and scheduled per level of the tree, there is no barrier in their execution. Each compositing step can be performed once the initiating block has rendered its image and all compositing steps scheduled with him as receiver have been completed. This allows the tree to be worked off in any order and in parallel as long as these dependencies are met.

Furthermore, each node can determine the structure of the compositing tree completely on its own and inform all local blocks appropriately about their roles as initators or receivers, thus completely avoiding costly network communication for coordination.

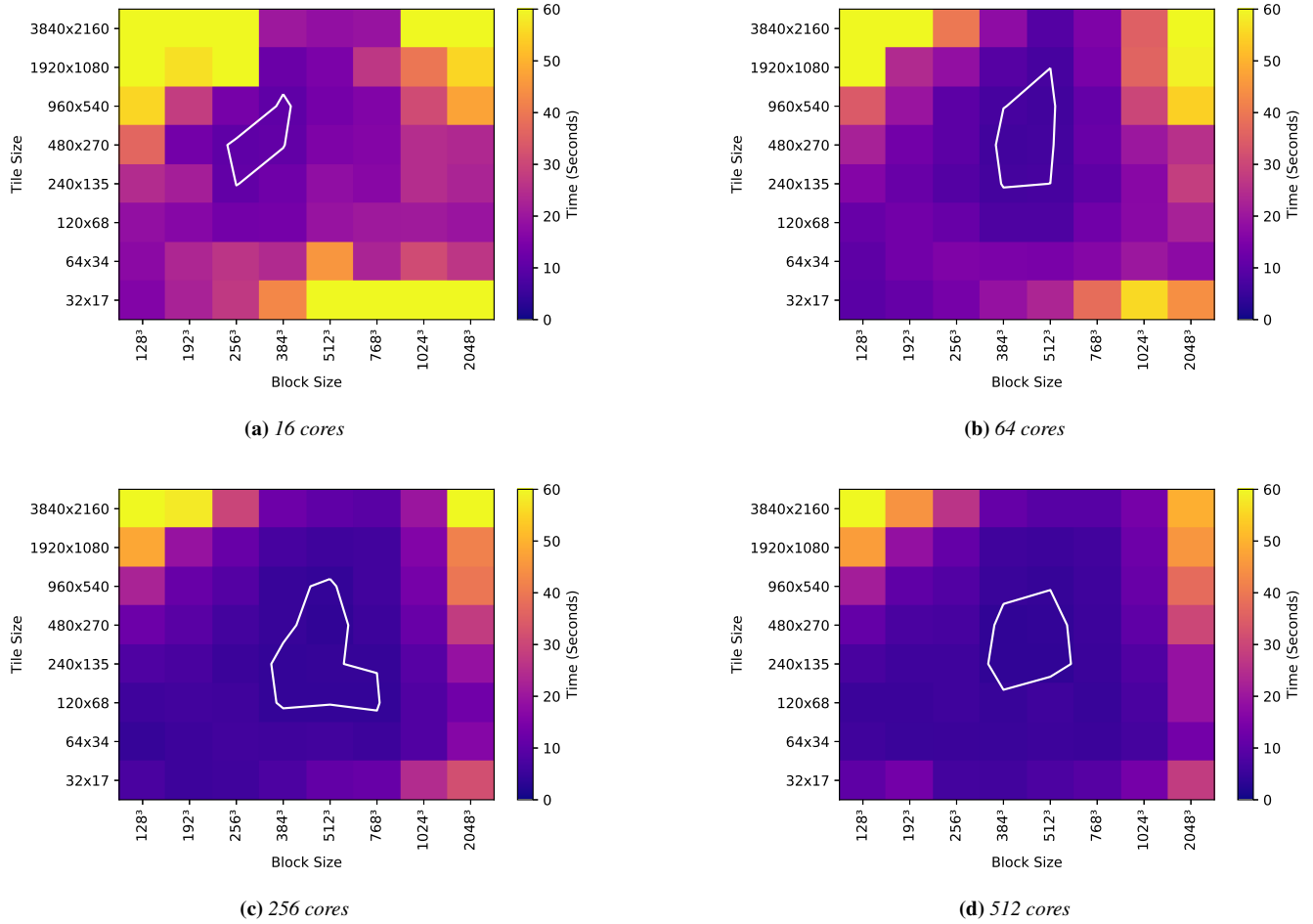
3.3. Optimization

Note that a block for which two compositing steps are scheduled with him being the receiver can merge the two images he receives even before his own rendering task is done, since the compositing steps are associative. However, since the calculation of the compositing steps is performed redundantly on each node and in parallel, one node might finish this calculation for a certain tile long before others. This would require to buffer incoming images for receiving blocks in the nodes lagging behind until they have scheduled the corresponding compositing steps.

To circumvent this buffering, the initiating block of each compositing step not only sends its image for compositing to the receiving block, but also the number of compositing steps the receiver should execute before dealing with the current step. This allows receiving blocks to merge multiple received images before their own image is ready and even before the compositing steps are calculated on its node. Consequently, the successful completion of these local operations is now only necessary to initiate a compositing step, not for being the receiver of one. This sender-initiated approach allows to perform each compositing operation as early as possible.

Additionally, we have implemented two common acceleration techniques for volume rendering: empty-block skipping and early ray termination. While empty-block skipping can be trivially integrated as a data preprocessing step during block initialization, it should be noted that there is a strong relationship to the optimal block size, as smaller blocks are more likely to be completely empty and can be skipped in compositing.

In constrast to early ray termination in standard shared-memory ray casting, our distributed tile-based compositing scheme performs opacity culling at tile granularity. After each compositing operation, the resulting image is checked against a predefined opacity threshold. Whenever the opacity of all fragments is saturated,



(a) *16 cores* **(b)** *64 cores*

(c) *256 cores* **(d)** *512 cores*

Figure 4: *Mean rendering times for different task granularities at 16, 64, 256 and 512 cores (*2048^3*,* 3250^3*,* 5160^3 *and* 6501^3 *voxels, respectively) in the weak in-situ scenario. The sweet spots are indicated as white contour lines with a threshold of 0.5 seconds around the best rendering time. Optimal performance is achieved in the middle ranges of both block and tile size parameters, with severe performance penalties in the extreme corner cases. The optimal configuration shifts slightly towards smaller tile sizes and larger block sizes as the number of cores increases.*

all outstanding blend-under compositing operations can be skipped and the tile can be immediately forwarded in the compositing tree. The corresponding superfluous rendering tasks that have not been started yet can be removed from the scheduling system, whereas the resulting images of already executing rendering tasks will simply be ignored.

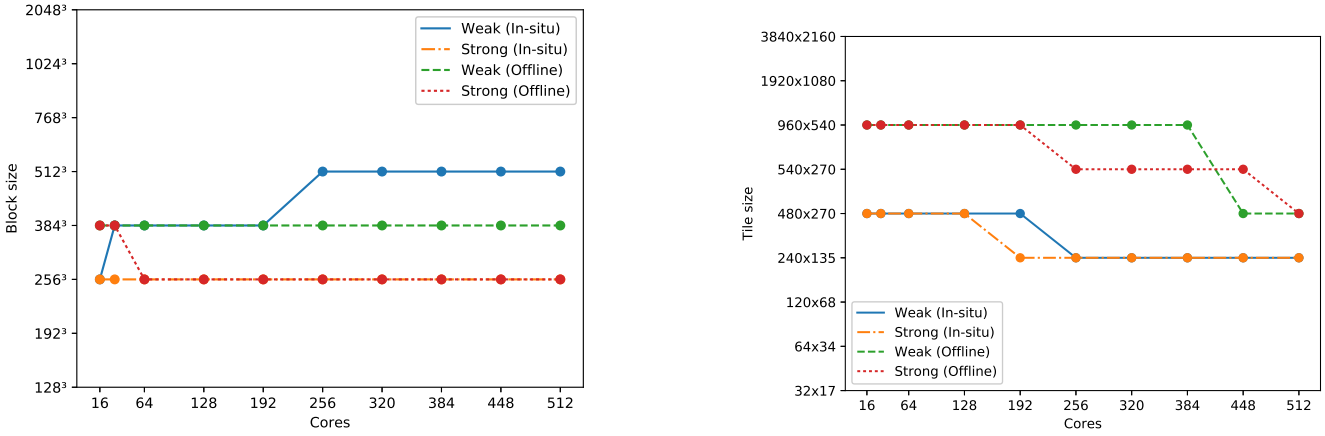
To assist early opacity culling, we have implemented a custom priority queue task scheduler, where pending block rendering tasks are dynamically kept sorted based on their Manhattan distance to the camera. This ensures full compute occupancy at any time while improving the execution order with respect to opacity culling as more rendering tasks are scheduled.

Figure 2 illustrates an exemplary compositing tree pattern and possible task execution timeline on two nodes for the scenario presented in Figure 1, featuring both empty-block skipping and opacity culling.

4. Implementation

Our novel task-based distributed rendering approach is based on the HPX (High Performance ParalleX) framework [KHAL*14], an aspiring task-based runtime environment with means for asynchronous communication across nodes. Each block is represented as an individual component in the active global address space (AGAS), allowing blocks to directly communicate in the compositing pattern. The custom priority queue scheduler is implemented on top of HPX's standard FIFO scheduler by manually keeping track of the number of rendering and compositing tasks being executed by HPX.

OSPRay [WJA*17] is used as rendering backend with its default internal TBB-based parallelization being disabled. A seperate scientific visualization renderer is instantiated for each CPU core, so all scheduled rendering tasks can render concurrently. Each block aggregates a shared structured volume instance and a pre-



(a) *Optimal block size*

(b) *Optimal tile size*

Figure 5: *Optimal block and tile sizes for up to 512 cores in the in-situ and offline scenario (both weak and strong scaling). The ideal block count appears to be independent from the number of cores. However, in offline rendering the block size additionally influences I/O performance. In general, the optimal tile size decreases with the number of cores. In the offline scenario larger tiles are beneficial.*

committed model instance, which is set as active model in the respective executing renderer.

Manual AVX2 intrinsics are used for standard blend-over image compositing, which allows to perform vectorized instructions on 8 consecutive RGBA pixels with 8 bit per channel. The same degree of vectorization was not achievable by relying on compiler-generated auto-vectorized code.

5. Results

To investigate the characteristics and potential of our novel task-based rendering system we have conducted comprehensive benchmarks with respect to optimal task granularity, task scheduling and scaling.

The *jet* data set (see Figure 3), which results from a direct numerical simulation of a jet of high-velocity fluid entering a medium at rest, was used with a standard fire and ice transfer function, thus yielding empty, transparent and fully opaque image areas. As reference configuration the data set was resampled to 2048^3 voxels (32 GB) on a single node. Timings were measured by rendering a full rotation around the data set at 3840x2160 resolution and computing the mean. The camera distance is adjusted to view the complete volume.

We have identified four important base scenarios to focus on: *in-situ* vs. *offline* and *weak* vs. *strong* scaling up to 512 cores. The in-situ scenario assumes block data is already in memory (e.g. after a preceding simulation run), whereas offline rendering requires additional on-demand I/O to load blocks into memory. For weak scaling, the total data size is upscaled proportionally to the number of cores, e.g. 6502^3 for 512 cores (approx. 275 gigavoxels). Strong scaling keeps the data size constant while increasing the number of cores.

All benchmarks were performed on the *Elwetritsch* cluster providing two Intel E5-2637v3 CPUs (16 cores) per node, 64GB of main memory and InfiniBand QDR interconnect.

5.1. Task Granularity

The granularity of the individual tasks in a task-based system crucially defines and limits the degree of possible parallelization and scalability. In our approach, we have two parameters to control task granularity: block size and tile size.

Figure 4 illustrates the mean rendering times across the full spectrum of block and tile sizes for 16, 64, 256 and 512 cores in the weak in-situ scenario, thus focusing on pure rendering performance without I/O. The sweet spots are clearly indicated in the middle ranges of both block and tile size parameters, with severe performance penalties in the extreme corner cases. The optimal configuration shifts slightly towards smaller tile sizes and larger block sizes as the number of cores increases.

The optimal block and tile sizes for all four scenarios across different core numbers is depicted in Figure 5. The ideal block count appears to be independent from the number of cores, in both weak and strong scaling scenarios. However, in offline rendering the block size additionally influences I/O performance. In general, the optimal tile size decreases with the number of cores as the increased fine-grained subdivision promotes latency hiding by task overlapping. Interestingly, in the offline scenario larger tiles are beneficial. A possible explanation is that larger tiles improve the early scheduling of on-demand I/O tasks.

5.2. Scheduling

Figure 6 depicts the scheduling of the individual I/O, rendering and compositing tasks in the offline scenario onto the 16 available cores of a single node. The diagram is representative for both the single and multi node cases.

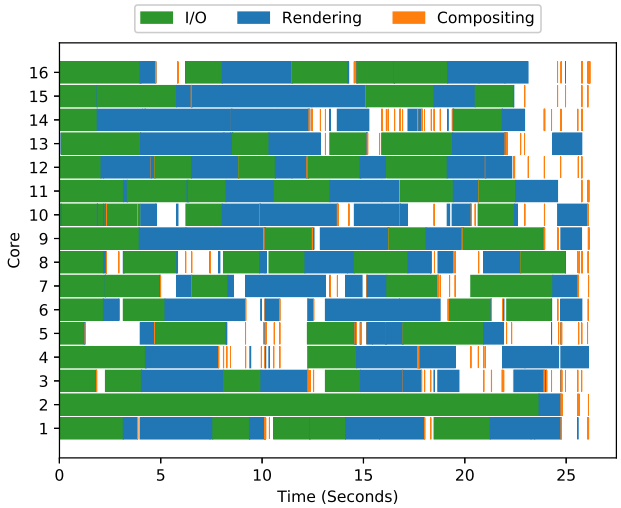


Figure 6: *Task scheduling for offline rendering of* 2048^3 *voxels using 16 cores on a single node at block size* 512^3 *and tile size 960x540. Block loading, rendering and compositing happen interleaved without barriers, so the resulting image is immediately ready after the final rendering and compositing tasks. Whitespace indicates unmet task dependencies such as outstanding I/O tasks or compositing partners.*

Clearly, there are no barriers in our task-parallel approach. Block loading, rendering and compositing happen interleaved, so the resulting image is immediately ready after the final rendering and compositing tasks. Especially the latency of costly I/O is hidden by automatic overlapping with computational tasks. The compositing tasks themselves are rather cheap in comparison to I/O and rendering.

5.3. Scaling

We have studied both weak and strong scaling characteristics of our approach in the in-situ and offline scenarios, as depicted in Figure 7. Note that in these benchmarks scaling only refers to the data size per node. However, for each benchmark the camera is adjusted such that the complete volume rendering is visible and its image area stays constant, thereby reducing the contribution of each node to the final image at bigger node counts.

After an initial performance improvement, weak scaling shows for both in-situ and offline cases approximately constant runtime, which is near optimal. This initial improvement is explained by the quick reduction in image contribution (i.e. rays) per node. Strong scaling rendering times in the offline scenario drop rapidly as the I/O overhead is distributed across nodes.

In general, strong scaling seems to be relatively limited in the tested scenarios. However, weak scaling suggests that strong scaling would improve at bigger workloads.

6. Conclusion and Outlook

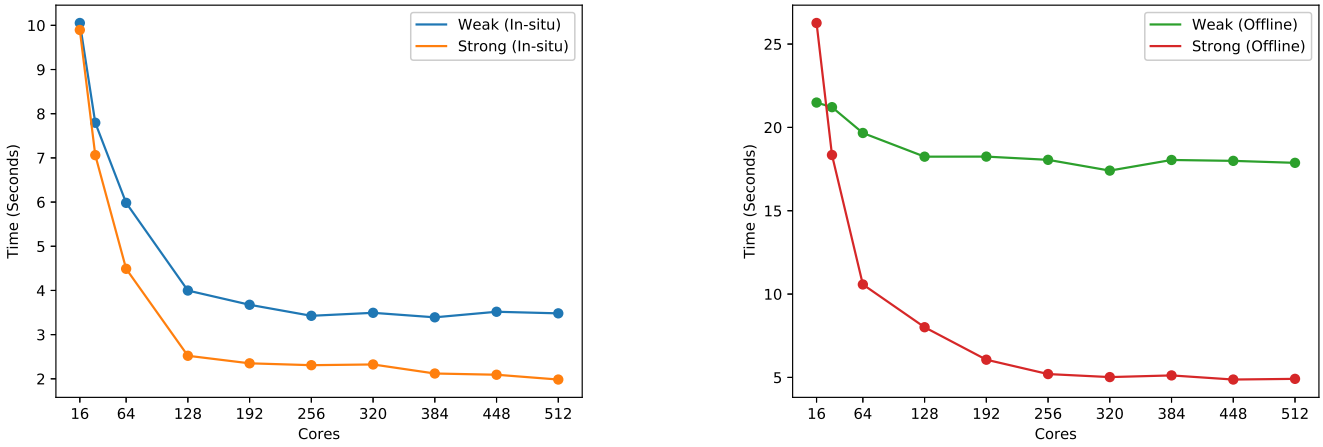
We have demonstrated a novel approach to large scale volume rendering based on distributed task-based runtime environments, an emerging trend in modern high performance computing on increasingly heterogeneous architectures. Our technique is based on a hybrid task-definition using parallelization in both image and data space, representing an effective and easy-to-control trade-off between sort-first and sort-last image compositing.

In our distributed asynchronous compositing scheme, each node determines the set of visible blocks for each tile. After an initial topological sorting, a sender-initiated binary tree communication scheme is used to correctly compose all block images within a tile. The compositing scheme incorporates common optimization techniques such as empty-block skipping and opacity culling, which is aided by a custom task priority scheduler based on the Manhattan distance to the camera.

We have conducted comprehensive benchmarks to study the characteristics of possible block and tile configurations in order to achieve optimal task granularity. The employed asynchronous binary tree compositing scheme enables good scalability in combination with inherent dynamic load balancing. The dynamic scheduling of initialization, rendering and compositing tasks on a single node ensures good latency hiding of network communication and I/O access.

Our contribution is intended as a baseline investigation of the applicability of task-based runtime environments to distributed scientific visualization. We anticipate that many enhancements and improvements of our approach are possible:

- Distributed work stealing would be an interesting approach to implement proper load balancing across node boundaries. Since each block is represented by an individual component in HPX's active global address space, the migration of blocks could be performed transparently with little to no modifications to the distributed compositing scheme. Distributed load balancing is especially important in an interactive setup with user-controlled camera navigation.
- In conjunction to distributed work stealing, a more sophisticated scheduling algorithm could also be used to improve task overlap and ensure the available I/O bandwidth is always kept saturated while executing rendering tasks as long as there are still blocks left to be loaded.
- Out of core handling could be used to support larger block counts on individual nodes.
- So far, our approach relies on the regular structure of blocks at various places. Support for unstructured data would involve complex enhancements to the distributed compositing scheme.
- In general, performance benchmarks on larger core counts would be very interesting. The integration of additional accelerator cards such as Intel Xeon Phi or GPGPUs is theoretically easy in the spirit of task-based runtime environments, but in practice still technically challenging.
- Additional benchmarks for comparison against traditional non-task-based approaches are required to further characterize the benefits and drawbacks of the presented approach. We do not expect significiant benefits from applications that already



(a) *In-situ scaling* **(b)** *Offline scaling*

Figure 7: *Weak and strong scaling up to 512 cores for both in-situ and offline scenarios. The corresponding block and tile sizes are depicted in Figure 5. After an initial performance improvement, weak scaling shows for both in-situ and offline cases approximately constant runtime. Strong scaling rendering times in the offline scenario drop rapidly as the I/O overhead is distributed across nodes.*

scale well on large machines using traditional data parallel approaches, especially if they are highly tuned and optimized towards a specific system or architecture. However, as elaborated in Section 1, we believe the major promising advantages of task-based designs lie in their portability to diverse and heterogeneous architectures, as well as the conceptionally more straightforward formulation of massively parallel programs.

- Besides distributed rendering, other load-sensitive techniques from scientific visualization such as topological methods or integral curve computations would certainly make promising candidates for task-based parallelization.
- Once enough task-based designs of standard visualization algorithms exist, their interplay and dependencies in a (complex) visualization pipeline could be studied.
- The task graph could be used for theoretical models and estimates about runtime, possible parallelization and scalability.

We will investigate these possibilities in future work.

Acknowledgement

This research was funded in part by the German Research Foundation (DFG) within the IRTG 2057 "Physical Modeling for Virtual Manufacturing Systems and Processes".

References

[AA07] AYYUB S., ABRAMSON D.: GridRod: A Dynamic Runtime Scheduler for Grid Workflows. In *Proceedings of the 21st Annual International Conference on Supercomputing* (New York, NY, USA, 2007), ICS '07, ACM, pp. 43–52. doi:10.1145/1274971.1274980. 3

[AGJ*14] ACUN B., GUPTA A., JAIN N., LANGER A., MENON H., MIKIDA E., NI X., ROBSON M., SUN Y., TOTONI E., WESOLOWSKI L., KALE L.: Parallel programming with migratable objects: Charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Piscataway, NJ, USA, 2014), SC '14, IEEE Press, pp. 647–658. doi:10.1109/SC.2014.58. 3

[BHP15] BEYER J., HADWIGER M., PFISTER H.: State-of-the-Art in GPU-Based Large-Scale Volume Visualization. *Computer Graphics Forum 34*, 8 (2015), 13–37. doi:10.1111/cgf.12605. 2

[BHPB03] BETHEL E. W., HUMPHREYS G., PAUL B., BREDERSON J. D.: Sort-First, Distributed Memory Parallel Visualization and Rendering. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (Washington, DC, USA, Oct 2003), PVG '03, pp. 41–50. doi:10.1109/PVGS.2003.1249041. 2

[BTSA12] BAUER M., TREICHLER S., SLAUGHTER E., AIKEN A.: Legion: Expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for* (Nov 2012), pp. 1–11. doi:10.1109/SC.2012.71. 3

[CDM06] CHILDS H., DUCHAINEAU M., MA K.-L.: A scalable, hybrid scheme for volume rendering massive data sets. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2006), EGPGV '06, Eurographics Association, pp. 153–161. doi:10.2312/EGPGV/EGPGV06/153-161. 2

[CM93] CORRIE B., MACKERRAS P.: Parallel volume rendering and data coherence. In *Proceedings of 1993 IEEE Parallel Rendering Symposium* (Oct 1993), pp. 23–26, 106. doi:10.1109/PRS.1993.586081. 2

[DCH88] DREBIN R. A., CARPENTER L., HANRAHAN P.: Volume Rendering. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1988), SIGGRAPH '88, ACM, pp. 65–74. doi:10.1145/54852.378484. 2

[DG15] DUBEY A., GRAVES D. T.: *A Design Proposal for a Next Generation Scientific Software Framework*. Springer International Publishing, Cham, 2015, pp. 221–232. doi:10.1007/978-3-319-27308-2_19. 1, 3

[DLS*09] DINAN J., LARKINS D. B., SADAYAPPAN P., KRISHNAMOORTHY S., NIEPLOCHA J.: Scalable Work Stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 53:1–53:11. doi:10.1145/1654059.1654113. 1

[EP07] EILEMANN S., PAJAROLA R.: Direct Send Compositing for Parallel Sort-last Rendering. In *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, 2007), EGPGV '07, pp. 29–36. doi:10.2312/EGPGV/EGPGV07/029-036. 2

[FCS*10] FOGAL T., CHILDS H., SHANKAR S., KRÃIJGER J., BERGERON R. D., HATCHER P.: Large Data Visualization on Distributed Memory Multi-GPU Clusters. In *High Performance Graphics* (2010), Doggett M., Laine S., Hunt W., (Eds.), HPG '10, pp. 57–66. doi:10.2312/EGGH/HPG10/057-066. 2

[GABS13] GILMANOV T., ANDERSON M., BRODOWICZ M., STERLING T.: Application characteristics of many-tasking execution models. In *The 19th International Conference on Parallel and Distributed Processing Techniques and Applications* (Las Vegas, USA, July 2013). 3

[HBC12] HOWISON M., BETHEL E. W., CHILDS H.: Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems. *IEEE Transactions on Visualization and Computer Graphics 18*, 1 (Jan 2012), 17–29. doi:10.1109/TVCG.2011.24. 1, 2

[HLYD11] HAIDAR A., LTAIEF H., YARKHAN A., DONGARRA J.: Analysis of Dynamically Scheduled Tile Algorithms for Dense Linear Algebra on Multicore Architectures. *Concurr. Comput. : Pract. Exper. 24*, 3 (Mar. 2011), 305–321. doi:10.1002/cpe.1829. 3

[KBM*08] KALE L. V., BOHM E., MENDES C. L., WILMARTH T., ZHENG G.: Programming Petascale Applications with Charm++ and AMPI. In *Petascale Computing: Algorithms and Applications*, Bader D., (Ed.). Chapman & Hall / CRC Press, 2008, pp. 421–441. 3

[KHAL*14] KAISER H., HELLER T., ADELSTEIN-LELBACH B., SERIO A., FEY D.: HPX: A Task Based Programming Model in a Global Address Space. In *Proc. of the 8th International Conference on Partitioned Global Address Space Programming Models* (New York, NY, USA, 2014), PGAS '14, ACM, pp. 6:1–6:11. doi:10.1145/2676870.2676883. 1, 3, 5

[KPH*10] KENDALL W., PETERKA T., HUANG J., SHEN H.-W., ROSS R.: Accelerating and Benchmarking Radix-k Image Compositing at Large Scale. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, 2010), EG PGV'10, pp. 101–110. doi:10.2312/EGPGV/EGPGV10/101-110. 2

[KWN*14] KNOLL A., WALD I., NAVRATIL P., BOWEN A., REDA K., PAPKA M. E., GAITHER K.: RBF Volume Ray Casting on Multicore and Manycore CPUs. In *Proceedings of the 16th Eurographics Conference on Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2014), EuroVis '14, Eurographics Association, pp. 71–80. doi:10.1111/cgf.12363. 2

[Lev90] LEVOY M.: Efficient Ray Tracing of Volume Data. *ACM Trans. Graph. 9*, 3 (July 1990), 245–261. doi:10.1145/78964.78965. 2

[MAWM11] MOLONEY B., AMENT M., WEISKOPF D., MOLLER T.: Sort-First Parallel Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 17*, 8 (Aug. 2011), 1164–1177. doi:10.1109/TVCG.2010.116. 2

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl. 14*, 4 (July 1994), 23–32. doi:10.1109/38.291528. 2

[MMD06] MARCHESIN S., MONGENET C., DISCHLER J.-M.: Dynamic Load Balancing for Parallel Volume Rendering. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2006), EGPGV '06, Eurographics Association, pp. 43–50. doi:10.2312/EGPGV/EGPGV06/043-050. 2

[MPHK94] MA K.-L., PAINTER J. S., HANSEN C. D., KROGH M. F.: Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 59–68. doi:10.1109/38.291532. 2

[MSE06] MÜLLER C., STRENGERT M., ERTL T.: Optimized Volume Raycasting for Graphics-hardware-based Cluster Systems. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2006), EGPGV '06, Eurographics Association, pp. 59–67. doi:10.2312/EGPGV/EGPGV06/059-066. 2

[NFLM07] NAVRÁTIL P. A., FUSSELL D. S., LIN C., MARK W. R.: *Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization*. Tech. rep., 2007. 2

[NPS12] NOTZ P. K., PAWLOWSKI R. P., SUTHERLAND J. C.: Graph-Based Software Design for Managing Complexity and Enabling Concurrency in Multiphysics PDE Software. *ACM Trans. Math. Softw. 39*, 1 (Nov. 2012), 1:1–1:21. doi:10.1145/2382585.2382586. 3

[PBH*16] PÉBAY P. P., BENNETT J. C., HOLLMAN D. S., TREICHLER S., MCCORMICK P. S., SWEENEY C., KOLLA H., AIKEN A.: Towards asynchronous many-task in situ data analysis using legion. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016* (2016), pp. 1033–1037. doi:10.1109/IPDPSW.2016.24. 3

[PCG*09] PUGMIRE D., CHILDS H., GARTH C., AHERN S., WEBER G. H.: Scalable Computation of Streamlines on Very Large Datasets. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 16:1–16:12. doi:10.1145/1654059.1654076. 1

[PGR*09] PETERKA T., GOODELL D., ROSS R., SHEN H.-W., THAKUR R.: A Configurable Algorithm for Parallel Image-compositing Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 4:1–4:10. doi:10.1145/1654059.1654064. 2

[PYRM08] PETERKA T., YU H., ROSS R., MA K.-L.: Parallel Volume Rendering on the IBM Blue Gene/P. In *Eurographics Symposium on Parallel Graphics and Visualization* (2008), Favre J. M., Ma K.-L., (Eds.), The Eurographics Association. doi:10.2312/EGPGV/EGPGV08/073-080. 2

[SML*03] STOMPEL A., MA K.-L., LUM E. B., AHRENS J., PATCHETT J.: SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering. In *Proc. of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (Washington, DC, USA, 2003), PVG '03, IEEE Computer Society, pp. 6–. doi:10.1109/PVGS.2003.1249040. 2

[WJA*17] WALD I., JOHNSON G., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J., GÜNTHER J., NAVRATIL P.: OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017). doi:10.1109/TVCG.2016.2599041. 5

[YWM08] YU H., WANG C., MA K.-L.: Massively Parallel Volume Rendering Using 2-3 Swap Image Compositing. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 48:1–48:11. doi:10.1109/SC.2008.5219060. 2